Adaptive game AI

Adaptive Game AI

Adaptive Game AI

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Maastricht,
op gezag van de Rector Magnificus,
Prof. mr. G.P.M.F. Mols,
volgens het besluit van het College van Decanen,
in het openbaar te verdedigen
op vrijdag 20 mei 2005, om 12:00 uur

door

Pieter Hubert Marie Spronck

Promotores: Prof. dr. H.J. van den Herik
Prof. dr. E.O. Postma

Leden van de beoordelingscommissie:
Prof. dr. A.J. van Zanten (Universiteit Maastricht; voorzitter)
Prof. dr. P.M.E. de Bra (Eindhoven University of Technology)
Prof. ir. L.A.A.M. Coolen (Universiteit Maastricht)
Prof. dr. H.J.M. Peters (Universiteit Maastricht)
Prof. dr. J. Schaeffer (University of Alberta)

SIKS Dissertation Series No. 2005-06
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 90-5278-462-0
Universitaire Pers Maastricht

Printed by Datawyse b.v., Maastricht, The Netherlands.

Contents

Preface

> The world is governed more by appearance than by realities,
> so that it is fully as necessary
> to seem to know something as it is to know it.
> — Daniel Webster (1782–1852).

The classic Greek culture recognised six art forms: painting, sculpture, architecture, literature, drama, and music. In the twentieth century, television, cinema, and comic books became known as the seventh to ninth art forms. The brothers Le Diberer (1993) nominated commercial computer games[1] as the tenth art form.

Many people will scoff at the notion of games being elevated to the status of art. They see games as little more than running through dark corridors and shooting aliens on a computer, which hardly can be considered art. These people have a point. Most games are too shallow to be called art. But, as we may not expect every book to be GÖDEL, ESCHER, BACH, every movie to be CITIZEN KANE, or every piece of music to be the BRANDENBURGER CONCERTOS, we may not expect every game to be high art. Certainly a few games exist that evoke profound, emotionally touching, fascinating experiences. It is true that such games are extremely rare. However, games are a young art form; when they mature more games will be found worthy of the epithet 'art'.

Games are certainly distinct from the other nine art forms. For one thing, they are the only art form that, by definition, needs to be experienced interactively. For a game to be considered art, the interaction in particular must be successful, so that game players may become deeply immersed in a game world, gaining a suspension of disbelief (i.e., a mental willingness to accept the game world as reality). Unfortunately, a suspension of disbelief is fragile, and shatters easily. To maintain it, every aspect of the game world must be true to the nature it is supposed to embody.

Nowadays, a game's top-notch graphics and sound manage to keep up a suspension of disbelief quite well. However, the behaviours of characters in a game are usually of an inferior quality. It is all too clear that the characters are lifeless, mindless drones controlled by a computer with little knowledge.

A major distinguishing feature of real-life beings, which is clearly lacking in characters in today's games, is the ability to adapt to new situations. Endowing

[1] Henceforth, whenever I use the term 'game' without an adjective, I am referring to a 'commercial computer game'.

computer-controlled characters with this ability may evoke the illusion that the characters actually understand what they are doing, and thus maintain the suspension of disbelief for a longer time.

The behaviour of characters in a game is determined by the so-called 'game AI' (AI being the abbreviation of 'Artificial Intelligence'). This thesis discusses how game AI can be made adaptive. The research is mainly driven by the goal of achieving results that are practically applicable. The research may be considered successful if, in a few years time, the investigated techniques are implemented in actual commercially-available games.

I am deeply grateful to the Institute of Knowledge and Agent Technology (IKAT) of the Universiteit Maastricht, which allowed me to do my thesis research as part of my job. In authoring this thesis, it was my good fortune to benefit from the invaluable guidance of Jaap van den Herik and Eric Postma. I am thankful to Ida Sprinkhuizen-Kuyper, Sander Bakkes, and Marc Ponsen, for our productive collaboration on considerable chunks of my research. I also thank my colleagues at IKAT, for our pleasant and fruitful discussions. My 2003 visit to Edmonton, Canada, proved to be a turning point in my research, for which I wish to express my thanks to the University of Alberta's GAMES group, led by Jonathan Schaeffer, and to BioWare Corp. Finally, I wish to extend my heartfelt gratitude to my parents, for their continued support, and to Muriël and Myrthe, for joy and love.

Pieter Spronck, January 2005.

Chapter 1

Introduction

A great deal of intelligence can be invested in ignorance
when the need for illusion is deep.
— Saul Bellow (b. 1915).

Over the last twenty years the audiovisual qualities of commercial games have improved significantly. However, over the same period game developers have largely neglected artificial intelligence (AI) in games, so-called 'game AI'. Since the turn of the century game-development companies have discovered that nowadays it is the quality of game AI that distinguishes good games from mediocre ones. The general goal of the present thesis is to investigate to what extent the quality of game AI can be improved by using machine-learning techniques. In particular, the goal is to create game opponents that can learn from mistakes and that can adapt to new tactics.

This chapter implicitly provides my research motivation. Section 1.1 examines the differences between analytical and commercial games. Section 1.2 discusses the state of the art in commercial game AI. Section 1.3 establishes that game AI can benefit from being adaptive. Section 1.4 discusses the scientific relevance of adaptive-game-AI research. The problem statement that guides the research is formulated in Section 1.5, along with three research questions. The chapter ends with an outline of the thesis in Section 1.6.

1.1 Analytical vs. Commercial Games

Computer games can be roughly divided into two groups, namely 'analytical games' and 'commercial games'. Analytical games are the classic board and card games, such as Backgammon, Bridge, Checkers, Chess, Go, Poker, and Stratego. Commercial games are the popular modern computer games, of which well-known examples are Baldur's Gate, Doom, EverQuest, Pacman, Quake, Tomb Raider, and Warcraft.

Traditionally, computer-game research has focussed on analytical games. The goal of computer-game research is to endow computers with artificial intelligence that makes them the strongest possible game-players. For some games the research has achieved impressive results; for instance, computers outplay World Champions in CHESS (Hsu, 2002), CHECKERS (Schaeffer, 1997), and OTHELLO (Buro, 1997).

Around the start of the twenty-first century computer-game research was extended to encompass commercial games (Woodcock, 1999). A close inspection shows that analytical and commercial computer games[1] differ in many characteristics. Nine of those differences are listed here.

Game-theoretical classification: Game theory distinguishes between perfect and imperfect information games, as well as between deterministic and stochastic games (Koller and Pfeffer, 1997; Halck and Dahl, 1999). In perfect information games complete information on the state of the game is available, while in imperfect information games part of the game state is hidden. Deterministic games have no element of chance, while in stochastic games chance plays a prominent role. Figure 1.1 presents a coarse personal assessment of how some typical example games (both analytical and commercial) can be qualified according to these characteristics. As can be observed, in general, analytical games deal with much or even perfect information and are highly deterministic, while commercial games deal with little information and are highly stochastic (Buro, 2004; Chan *et al.*, 2004).[2]

Origin of complexity: The complexity of an analytical game arises from the interaction of a few simple, transparent rules. The complexity of a commercial game arises from the interaction of large numbers of in-game objects and locations, controlled by complex, opaque rules (Fairclough, Fagan, MacNamee, and Cunningham, 2001; Nareyek, 2002; Buro, 2004).

Computer requirement: Analytical games can, in principle, be played by humans without the use of a computer. Commercial games take place in a virtual world created by the computer, which means that the computer is an essential part of the game.

Pacing: Analytical games usually progress at a slow pace, while commercial games are fast-paced (Nareyek, 2002).

---

[1] The term 'commercial games' is misleading, because analytical games can be commercially exploited as well. An alternate term found in literature is 'interactive computer games', but since *all* computer games are interactive, this term is even more misleading. A potentially better term is 'video games', but this term is usually reserved for 'console games' that are played on dedicated gaming hardware connected to a television set. Most authors simply refer to commercial games as 'computer games' or 'games', and let the context define which type of games they are referring to. In this thesis I will use the simple term 'games' to refer to commercial computer games, except where I am discussing differences between analytical and commercial games, as in the present section.

[2] In imperfect-information analytical games little information is hidden, at least in comparison with commercial games. For instance, in card games only the players' hands are hidden, while in commercial games complete game worlds are hidden.

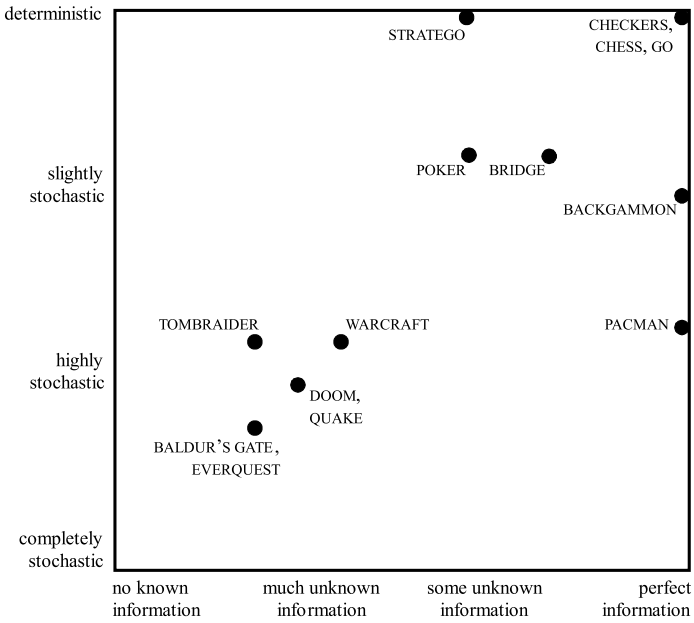

Figure 1.1: Game-theoretical classification of some analytical and commercial games according to the author. The horizontal axis represents the amount of information on the game state available to the player, while the vertical axis represents the amount of randomness in the game.

Drama: The only drama in connection with an analytical game is the drama of winning or losing. For most commercial games drama, in the form of a story (however shallow), is an essential part of the game (Laurel, 1993).

Role reversal: In analytical games the computer replaces one or more of the human players. In essence, the computer transcends into the human world to assume the role of a game-playing human. In commercial games human players take on the role of some of the virtual characters in the game (whether those characters are actual beings in the game, or god-like army leaders that have no in-game avatar) – the human player becomes part of the computer world.

Player skills: Analytical games require players to use first and foremost their intellectual skills.[3] Commercial games require players to invest a wide variety of skills. Depending on the game, besides intellectual skills players will need to use their imagination, reflexes, timing skills, sensory abilities, emotions, and even ethical insights.

[3] In analytical games between humans, usually psychology also plays an important role. However, in an analytical game played between a human and a computer, psychology is not used as a strategic means, at least not yet.

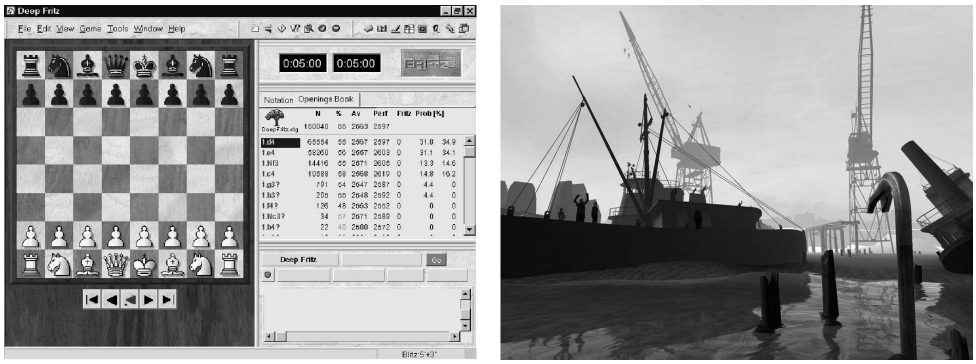

Figure 1.2: The difference in art between a typical analytical game (DEEP FRITZ, left) and a typical commercial game (HALF-LIFE 2, right).

Art: For analytical games the art, consisting of graphics and sound, is of little importance. For commercial games art is of key importance. Most of the development resources of a commercial game are invested in the game's art (Fairclough *et al.*, 2001; Khoo and Zubek, 2002). This difference is vividly illustrated in Figure 1.2.

Goal: For analytical games the goal of the computer is to defeat the human player. For commercial games the goal of the computer is to entertain the human player (Tozour, 2002b; Chan *et al.*, 2004; Lidén, 2004).

Decades of research (often very successful) have been invested into AI that plays analytical games (Schaeffer and Van den Herik, 2002; Van den Herik, Uiterwijk, and Van Rijswijck, 2002; Van den Herik, Iida, and Heinz, 2003). The vast majority of this research focusses on deterministic, perfect information games (Halck and Dahl, 1999). The aforementioned differences between analytical games and commercial games are a reason that most analytical-game research has little applicability to commercial games. There are many problems in the field of commercial-game AI that are untouched by analytical game research, such as pathfinding, spatial and temporal reasoning, and decision making under high uncertainty (Buro, 2003b).

This thesis investigates commercial-game AI. The research has little overlap with analytical game research. Henceforth, the term 'game' will be used to refer to a 'commercial computer game'.

1.2 Game AI

The popularity surge of commercial games has stimulated the growth of the game-development industry until its revenues surpassed those of the Hollywood movie industry (Hause, 1999; Fairclough *et al.*, 2001; Snider, 2002). Traditionally, game-

development companies competed by creating games with superior graphics. Nowadays they attempt to compete by offering a better game-play experience (Tozour, 2002b; Graepel, Herbrich, and Gold, 2004). The behaviour of game characters is an essential element of game-play. Game AI is defined as the decision-making algorithms of game characters, that determine the characters' behaviour (Wright and Marshall, 2000; Allen *et al.*, 2001; Fairclough *et al.*, 2001; Nareyek, 2002). Game AI has become an important selling point of games (Laird and Van Lent, 2001; Forbus and Laird, 2002). However, even state-of-the-art game AI is, in general, of low quality (Laird and Van Lent, 2001; Schaeffer, 2001; Buro, 2004; Gold, 2004). Game AI can benefit from academic research into commercial games (Forbus and Laird, 2002), although this research is still in its infancy (Laird and Van Lent, 2001).

It should be noted that the term 'game AI' is used differently by game developers and academic researchers (Funge, 2004; Gold, 2004; Nareyek, 2004). Academic researchers restrict the use of the term 'game AI' to refer to intelligent behaviours of game characters (Wright and Marshall, 2000; Allen *et al.*, 2001; Funge, 2004). In contrast, for game developers the term 'game AI' is used in a broader sense to encompass techniques such as pathfinding, animation systems, level geometry, collision physics, vehicle dynamics (Tomlinson, 2003) and even the generation of random numbers (Rabin, 2004a).

In this thesis the term 'game AI' will be used in the narrow, academic sense. Furthermore, the term 'agent' will be used to refer to any decision-making game presence, whether it is a 'visible' agent (e.g., a creature that attacks the player), or it is an 'invisible' agent (e.g., the commander of an army that opposes the player). The focus of this thesis lies on agents that compete with a human player. These agents are called 'opponents'.

In general, game AI may operate on three levels of intelligence, namely (i) operational, (ii) tactical, and (iii) strategic. On the operational level, game AI controls the movements and individual actions of an agent. On the tactical level, game AI determines sequences of actions for an agent to accomplish a specific goal in an environment. On the strategic level, game AI engages in long-term planning of decisions for an agent. This thesis discusses game AI at all three levels of intelligence.

The remainder of this section discusses the goals that game AI aims to achieve (1.2.1), and the state of the art in game AI (1.2.2).

1.2.1 Goals

The purpose of a game is to provide entertainment (Tozour, 2002b; Nareyek, 2004). By extension this is also the purpose of game AI. Thus, the question that is in the forefront of any game-AI programmer's mind is: "How can game AI contribute to a game's entertainment value?"

Most games pose a challenge to human players in the form of opponents, whose behaviour is controlled by game AI. Three important issues with respect to the entertainment value that opponents provide are the following. First, a challenge is *not* entertaining when it is too easy or too hard (Graepel *et al.*, 2004). Second, most human players who are defeated by a computer will be disappointed if they feel they

lost undeservedly. Third, human players generally appreciate an agent maintaining the illusion that it is really intelligent (Scott, 2002). Considering these three issues, the following is a (not necessarily exhaustive) list of seven goals, arranged according to increasing difficulty, that game AI aspires to for providing an entertaining challenge. The better game AI achieves the goals, the higher its quality.

No obvious cheating: An agent cheats when it uses information or executes actions that are in principle unavailable to the human player. For most games some form of cheating by game AI is unavoidable (Scott, 2002) and implemented deliberately. This is not necessarily a problem, as long as the cheating is not too obvious. In general, state-of-the-art games do not employ obvious cheating to create challenging opponents.

Unpredictable behaviour: An agent whose actions are predictable is usually easy to defeat (if not plain boring) and does not present an illusion of intelligence (Crawford, 1984). With random variations on manually designed behaviour unpredictable behaviour can be achieved easily. Unfortunately, with random variations game AI will not always be equally challenging. [4] Expert human players may prefer non-random behaviour, as long as it provides a strong challenge.

No obvious inferior behaviour: The moment an agent performs a clearly bone-headed action, the illusion of its intelligence is shattered (Crawford, 1984). Obvious inferior agent behaviour is often the result of programming mistakes that went undetected during a game's 'quality assurance' phase (Tozour, 2002a). Even state-of-the-art games do not succeed in avoiding such behaviour entirely.

Using the environment: Games are commonly situated in a virtual world, with a wealth of environmental features that can be tactically exploited. To allow agents to exploit them equally well as human players, some game developers let the game AI take environmental features into account. Usually, this is realised by adding markings to the environment (Lidén, 2002; Tomlinson, 2003; Orkin, 2004b), or by allowing the environmental features to communicate their possibilities to the game AI (Orkin, 2002, 2004a). One step further, game AI is able to explore and analyse a game world by itself to form new tactical plans. As yet, advanced game AI with such capabilities is only explored in academic research, e.g., by Laird (2001).

Self-correction: Far worse than an agent that makes an exploitable mistake, is an agent that consistently repeats the same mistake. To allow game AI to avoid the repetition of mistakes, it should be able to (i) recognise a mistake, and (ii) change the agent's behaviour to avoid the mistake in the future. The behaviour learning must take place 'online', i.e., while the game is being played, because game AI must learn from the mistakes it makes in actual game-play situations.

[4] In the source code of the game AI of version 1.31 of the game Neverwinter Nights the following change comment can be found, dated September 19, 2002: "Removed randomness from Talent system. You can't have smart AI and random behavior."

Furthermore, the learning must be unsupervised, because the human player cannot be expected to inform the game AI that a mistake was made. As yet, there is no precedent of the successful application of unsupervised online learning in mainstream top-rated games (Manslow, 2002; Kirby, 2004).

Creativity: Avoiding the repetition of mistakes usually can be achieved by changing parameters (e.g., reducing the occurrence rate of one action in favour of an other). When game AI is confronted with a previously unconsidered situation (e.g., the human player using a surprising new tactic), simple parameter changing will be of little help. The game AI must creatively learn completely new behaviour. For games, the most advanced form of adapting to new situations in practice is game AI that is allowed to choose between a limited number of predefined tactics (Johnson, 2004).

Human-like behaviour: Similar to the ultimate goal of any AI researcher, the ultimate goal of a game-AI designer is to create AI that rivals human intelligence. For games this is not an unreachable goal, because game worlds have a limited scope. However, it is obvious that human-like game behaviour is an advancement that can only be achieved after all other mentioned goals have been reached (Laird, 2001; Livingstone and McGlinchey, 2004).

1.2.2 State of the Art

Even in state-of-the-art games the game AI lacks sophistication. Of the seven game-AI goals listed in Subsection 1.2.1 only the first three are addressed by modern game AI – and often not successfully. The four main reasons for this low quality of game AI are the following (adopted from Fairclough *et al.*, 2001).

- The need for advanced graphics still overshadows the need for good game AI.
- Game-development companies and their publishers are distrustful of advanced AI techniques.
- Game AI is usually added when the deadline for the release of a game approaches, and there is little time left to experiment.
- Game developers commonly lack academic knowledge of AI.

To develop better game AI, game-development companies need help from the academic community (Laird and Van Lent, 2001; Rabin, 2004b). This thesis comprises an academic contribution to game-AI research. Its focus is on the the fifth and the sixth goal listed in Subsection 1.2.1: 'self-correction' and 'creativity' – in brief, its focus is on the investigation of 'adaptive game AI'.

1.3 Adaptive Game AI

Adaptive game AI is defined as game AI with the ability of self-correction (i.e., the ability to resolve faulty agent behaviour), and with the ability of creativity (i.e., the ability to adapt successfully to changing circumstances). Since there is no precedent for the use of adaptive game AI in state-of-the-art games, it should be considered carefully whether it is a good idea to enhance games with adaptive game AI. In this respect I will discuss the following three questions: (i) To what extent is adaptive game AI beneficial for games? (ii) Is adaptive game AI really necessary? and (iii) Can adaptive game AI contribute to the purpose of games: providing entertainment? These questions are answered in Subsections 1.3.1, 1.3.2, and 1.3.3, respectively.

1.3.1 Benefits

The answer to the question "To what extent is adaptive game AI beneficial for games?" is that adaptive game AI (i) allows the challenge level of a game to be maintained automatically, and (ii) improves the effectiveness of the 'quality assurance' phase of game development.

To illustrate why maintenance of the challenge level of a game is beneficial, I provide as an example the game AI of the second game in the BALDUR'S GATE series: SHADOWS OF AMN. SHADOWS OF AMN is a so-called 'computer roleplaying game' (CRPG). In the game the player controls a team of agents who exist in a world where they meet many enemies. Among the toughest enemy types are dragons (illustrated in Figure 1.3). According to CRPG tradition, dragons are both physically and mentally powerful creatures. While SHADOWS OF AMN does not require the player to fight dragons, the designers realised that most players will attempt to do so anyway. Therefore they created complex game AI that should be able to humiliate any player bold enough to attack a dragon. Soon after the game's release, weaknesses in the game AI were discovered that players could exploit to defeat any dragon in the game, even with a weak team.[5] Furthermore, without exploiting game AI weaknesses, players could still design superior tactics that, while unforeseen by the game developers, allowed weak teams to take on dragons successfully. It is trivial for a dragon to recognise that its current behaviour is inadequate to deal with tactics used by attackers that, according to its domain knowledge, are no match for it. Were the dragons controlled by adaptive game AI instead of static game AI, an answer to the superior and exploiting tactics could have been discovered automatically, keeping up the challenge level of the game.

During the 'quality assurance' phase of game development, adaptive game AI can be used to spot weaknesses in manually-designed game AI, and to suggest alternative tactics. This application of adaptive game AI is an inexpensive investment that has

[5] One of these exploits was that dragons only responded to visible attackers. As long as the attackers remained outside the visual range of a dragon while attacking, it would not fight back. A second exploit was that the player team could lay traps all around a dragon, that killed it as soon as they went off. A dragon would not interfere with laying traps, even though it obviously is a hostile action. These exploits were fixed in an add-on to the game that appeared one year after the initial release.

Figure 1.3: A surprisingly meagre challenge: a dragon in SHADOWS OF AMN.

the potential to deliver valuable results, risk-free (Spronck, Sprinkhuizen-Kuyper, and Postma, 2002; Chan *et al.*, 2004). Even if game developers and publishers are hesitant to incorporate adaptive game AI in their games (which they are), they can still apply adaptive game AI during the 'quality assurance' phase.

1.3.2 Necessity

The answer to the question "Is adaptive game AI really necessary?" is that adaptive game AI is sorely needed to deal with the complexities of state-of-the-art games.

Over the years games have become increasingly complex, offering realistic worlds, freedom and a great variety of possibilities. The technique of choice used by game developers for dealing with a game's complexities is rule-based game AI, usually in the form of scripts (Nareyek, 2002; Tozour, 2002c). The advantage of the use of scripts is that scripts are (i) understandable, (ii) predictable, (iii) tuneable to specific circumstances, (iv) easy to implement, (v) easily extendable, and (vi) useable by non-programmers (Tozour, 2002c; Tomlinson, 2003). However, as a consequence of game complexity, scripts tend to be quite long and complex (Brockington and Darrah,

2002). Manually-developed complex scripts are likely to contain design flaws and programming mistakes (Nareyek, 2002). Successful adaptive game AI can ensure that the impact of these mistakes is limited to only a few situations encountered by the player, after which their occurrence will have become unlikely. Consequently, it is safe to say that the more complex a game is, the greater the need for adaptive game AI (Fairclough *et al.*, 2001; Laird and Van Lent, 2001; Fyfe, 2004). In the near future game complexity will only increase. As long as the best approach to game AI is to design it manually, the need for adaptive game AI will increase accordingly.

1.3.3 Entertainment

The answer to the question "Can adaptive game AI contribute to the purpose of games: providing entertainment?" is that the capability of adaptive game AI to maintain the challenge level of a game positively influences the entertainment provided by a game (Crawford, 1984).

Game AI in most modern games is not challenging. The appeal of Massive Multiplayer Online Games (MMOGs), where human players challenge each other, stems partly from the fact that computer-controlled opponents often exhibit what has been called 'artificial stupidity' (Schaeffer, 2001) rather than artificial intelligence. Adaptive game AI has the potential to make the game AI more challenging, since it can learn automatically to defeat strong tactics used by the human player. Many researchers and game developers hold that game AI, in principle, is entertaining when it is difficult to defeat (Buro, 2003b).

Furthermore, adaptive game AI, if implemented correctly, cannot only be used to make the game AI stronger, but also to scale automatically the challenge level of the game AI to the skills of the human player. On the subject of game AI challenges and entertainment, in his famous novel "2001: A Space Odyssey", Clarke (1968) writes about the artificially intelligent computer HAL 9000:

> "For relaxation [the astronauts] could always engage HAL in a large number of semi-mathematical games, including checkers, chess, and polyominoes. If HAL went all out, he could win anyone of them; but that would be bad for morale. So he had been programmed to win only fifty percent of the time, and his human partners pretended not to know this."

While it might be questioned whether adults are entertained when they win a game while knowing their opponent made deliberate mistakes, Clarke assumes correctly that humans, in general, will neither play a game when they know they just will be slaughtered, nor enjoy a game when they know their opponent is no match for them. The most enjoyable games are those that are played between opponents with a comparative level of skill (Graepel *et al.*, 2004). Therefore, if adaptive game AI continuously scales a game's difficulty level to the point that the human player is challenged, but not completely overpowered, the game will be most entertaining, and will remain entertaining even if the player's skill increases through experience.

1.4 Scientific Relevance

While games are generally considered to be a worthwhile research subject for social and cultural scientists, they may leave the impression to be too frivolous an application for computer scientists. This impression is misguided. Games are considered to be a driving force behind the research and development of 3D computer graphics and animation (Pabst, 2000; Philips-Mahoney, 2002; Sawyer, 2002). I argue that they are worthy of the same position for the research into artificial intelligence.

For artificial intelligence research, complex modern games are truly challenging applications. They have the following four characteristics.

- *Games are widely available.* AI innovations implemented in games are subjected to the scrutiny of hundreds of thousands of human players (Laird and Van Lent, 2001; Sawyer, 2002).
- *Games reflect the real world.* Games can often be considered simulations of aspects of reality. Therefore, game AI may capture features of real-world behaviour (Sawyer, 2002; Graepel *et al.*, 2004).
- *Games are a test-bed for human-like intelligence.* While 'real' human-like intelligence is not required for games, game AI must be able to simulate human-like behaviour to a large extent. Therefore, games are ideally suited to pursue the fundamental goal of AI, i.e., to understand and develop systems with human-like capabilities (Laird and Van Lent, 2001; Sawyer, 2002).
- *Games place highly-constricting requirements on implemented AI solutions.* Requirements for game AI force it to achieve good results with limited computational resources (Nareyek, 2002; Charles and Livingstone, 2004), free from possible degradation (Charles and Livingstone, 2004), in noisy environments (Laird and Van Lent, 2001), and within a few trials.[6]

By these characteristics, results achieved with game AI are widely applicable. They may be transferred to many other problem domains, which generally are less restrictive. Achieved results may contribute to, amongst others, the fields of machine learning, multi-agent systems, and robotics (Laird and Van Lent, 2001).

1.5 Problem Statement and Research Questions

Section 1.2 indicated that so far there is little academic research into commercial game AI. Section 1.3 indicated that adaptive game AI does not exist yet in state-of-the-art games. Furthermore, it is argued that adaptive game AI can be beneficial to games (1.3.1), that the need for adaptive game AI exists and will only increase in the near future (1.3.2), and that adaptive game AI can contribute to the purpose of games: providing entertainment (1.3.3).

[6] The requirements are further discussed in Subsection 2.3.4.

Successful adaptive game AI achieves the fifth and sixth goals listed for game AI (1.2.1), and thus contributes to the quality of game AI. The quality of game AI is directly related to its entertainment value (Tozour, 2002b). In this thesis it is assumed that machine-learning techniques can be used to implement adaptive game AI. Several research projects have investigated machine learning for game AI in simple games (Demasi and Cruz, 2002; Laramée, 2002a; Demasi and Cruz, 2003; McGlinchey, 2003). However, complex game AI (i.e., the game AI in complex games) so far is an untouched area.[7] Consequently, the problem statement discussed in this thesis reads as follows.

> **Problem statement**: To what extent can machine-learning techniques be used to increase the quality of complex game AI?

To find an answer to the problem statement, four research questions are formulated below.

For expert players adaptive game AI is successful if it increases the effectiveness of opponents, and thus their challenge level. Research into ways to implement effective adaptive game AI is related to research into the use of machine learning for agent control, such as evolutionary robotics (Arkin, 1998). In general, this research focusses on learning during the development phase of the control mechanism, so-called 'offline' learning. The first research question therefore reads as follows.

> **Research question 1**: To what extent can offline machine-learning techniques be used to increase the effectiveness of game AI?

While game AI can be improved by offline learning during game development, the actual confrontation with human players takes place during the deployment phase of a game. Game AI that adapts during the deployment phase of a game uses so-called 'online' learning. The second research question therefore reads as follows.

> **Research question 2**: To what extent can online machine-learning techniques be used to increase the effectiveness of game AI?

Most agent-AI research, both inside and outside the field of game research, aspires to make agents as effective as possible. In games, highly effective game AI is entertaining for expert human players. However, successful adaptive game AI should provide entertainment for all players, not just expert players. Novice players are entertained by game AI that matches their skill. Entertainment in games is best ensured if agents are challenging but not overpowering, against human players of all levels of skill. The third research question therefore reads as follows.

> **Research question 3**: To what extent can machine-learning techniques be used to scale the difficulty level of game AI to meet the human player's level of skill?

[7] At least, as far as unsupervised learning is concerned. Subsection 2.3.2 lists a few complex games with game AI that employs supervised learning.

Name	Type	AI level	Agents	Sections
Box-pushing	robot movement	operational	1	3.3
Food-gathering	search & avoid	operational	1	3.4
Duelling spaceships	RTS game	operational	1	4.1
QUAKE	action game	tactical	4	4.2
Simulated CRPG	CRPG	tactical	4	5.2–5.4
NEVERWINTER NIGHTS	CRPG	tactical	4	5.5
WARGUS	RTS game	strategic	> 50	6.2–6.4

Table 1.1: Game and game-like environments investigated in the thesis.

This thesis aims at providing a practical approach to the design and implementation of adaptive game AI. Consequently, it must consider how adaptive game AI is best applied by game-development companies. Hence, the fourth research question reads as follows.

> **Research question 4**: How can adaptive game AI be integrated in the game-development process of state-of-the-art games?

1.6 Thesis Outline

The thesis investigates seven different games and game-like environments. These are listed in Table 1.1, with their relevant characteristics. From left to right, the five columns of the table display (i) the environment's name, (ii) the environment's type (game types are discussed in Subsection 2.2.2), (iii) the level of intelligence on which the AI operates in the environment, (iv) the number of agents under the control of the AI, and (v) the thesis sections in which the environment is investigated.

The outline of this thesis is as follows.

Chapter 1 implicitly motivates the research, and formulates the problem statement and four research questions.

Chapter 2 provides background information. It presents (i) a short overview of the machine-learning techniques used in this thesis, (ii) an overview of the state of the art in game-AI research, and (iii) an exposition of the use of machine learning in game AI. It contributes to answering all research questions, in particular the second research question.

Chapter 3 contributes to answering the first research question. It presents a novel evolutionary technique called the 'Doping-driven Evolutionary Control Algorithm' (DECA). When evolving the behaviour of agents in game-like environments, DECA is able to achieve results that are more effective than results achieved with traditional evolutionary techniques. DECA is empirically validated by two experiments.

Chapter 4 contributes to answering both the first and second research questions. It investigates empirically to what extent evolutionary learning can be applied to improve game AI, both offline and online.

Chapter 5 contributes to answering the second, third, and fourth research questions. It presents a novel technique for online adaptation of game AI, called 'dynamic scripting'. The effectiveness of dynamic scripting is empirically confirmed in a game simulation and in an actual commercial game. It is also shown how dynamic scripting can be used to scale the game AI's difficulty level.

Chapter 6 contributes to answering the first, second, and fourth research questions. It discusses how offline adaptive game AI can be used to improve the reliability of online adaptive game AI, and how adaptive game AI can be integrated in the development process of modern games.

Chapter 7 first answers the four research questions and then comes to a conclusive answer to the problem statement. It finishes with several suggestions for future research.

Chapter 2

Background

In every real man a child is hidden that wants to play.
— Friedrich Wilhelm Nietzsche (1844–1900).

The focus of the present research is on the use of machine-learning techniques to improve the quality of game AI, specifically, to improve the decision-making capabilities of agents that compete with a human player. This chapter provides background information in support of the research, on three different subjects, namely machine-learning techniques in Section 2.1, games in Section 2.2, and the application of machine learning to game AI in Section 2.3. A summary of the chapter is provided in Section 2.4.

2.1 Machine Learning

This section provides a concise overview of the machine-learning techniques applied in the present research. It discusses evolutionary algorithms (2.1.1), artificial neural networks (2.1.2), evolutionary artificial neural networks (2.1.3), evolutionary control (2.1.4), and reinforcement learning (2.1.5).

2.1.1 Evolutionary Algorithms

'Biological evolution' (Dawkins, 1976, 1986) employs the theories of 'natural selection' (Darwin, 1859) and 'natural genetics' (Mendel, 1866) to explain how complex living beings, tuned to their environment, have come to exist. Evolutionary algorithms are search-and-optimisation algorithms based on the principles of biological evolution. The most widely known evolutionary algorithm is the 'genetic algorithm' (GA), developed by Holland (Holland, 1975; Goldberg, 1989; Bäck, 1996). Many other varieties of evolutionary algorithms have been invented, some of which are even older than genetic algorithms. Examples are evolution strategies (Schwefel, 1965; Bäck, 1996), evolutionary programming (Fogel, 1962; Bäck, 1996), clas-

sifier systems (Holland, 1975; Goldberg, 1989), and genetic programming (Koza, 1992; Kinnear, 1994). All evolutionary algorithms share the following five features.

- *Population*: Evolutionary algorithms optimise a collection of potential solutions to a problem, called a 'population'.
- *Chromosomes*: Evolutionary algorithms encode the potential solutions. The encoded solutions are called 'chromosomes'.
- *Fitness function*: Evolutionary algorithms assign each chromosome in the population a 'fitness' value, that indicates how well the potential solution encoded in the chromosome solves the problem, compared with the other potential solutions in the population.
- *Genetic operators*: To create new chromosomes, evolutionary algorithms apply transformation methods, called 'genetic operators', to 'parent' chromosomes, already existing in the population.
- *Selection*: To select parent chromosomes, evolutionary algorithms apply a selection mechanism to the population, which gives the fittest chromosomes the highest chance to procreate.

The idea is that an algorithm possessing these features will produce potential solutions that have a high chance of containing characteristics of well-working solutions. As long as the population has not converged too much, an evolutionary algorithm has the ability to escape from local optima. Arguably the most important property of evolutionary algorithms is that the only requirement for applying them is the ability to define an adequate fitness function. The main disadvantage of evolutionary algorithms is that they are not guaranteed to find a good solution, not even a mediocre one (Goldberg, 1989).

Genetic operators can be divided in three types, namely (i) reproduction operators, that create a child chromosome by copying a parent chromosome, (ii) mutation operators, that create a child chromosome by copying a parent chromosome and making changes to it, and (iii) crossover operators (also called 'recombination operators'), which combine chromosome parts of two or more parent chromosomes to create a child chromosome.

Each of the aforementioned varieties of evolutionary algorithms prescribes specific implementations of chromosome encoding, genetic operators, selection, and other parameters. Nowadays researchers are unlikely to follow the prescriptions, but use whatever they think fits best to the problem which they attempt to solve. The researchers refer to their algorithm with the umbrella name 'evolutionary algorithm'.

Evolutionary algorithms are employed in Chapters 3, 4, and 6.

2.1.2 Artificial Neural Networks

Artificial neural networks, also called simply 'neural networks', are structures that can learn to emulate a (non-linear) function. A neural network consists of a network

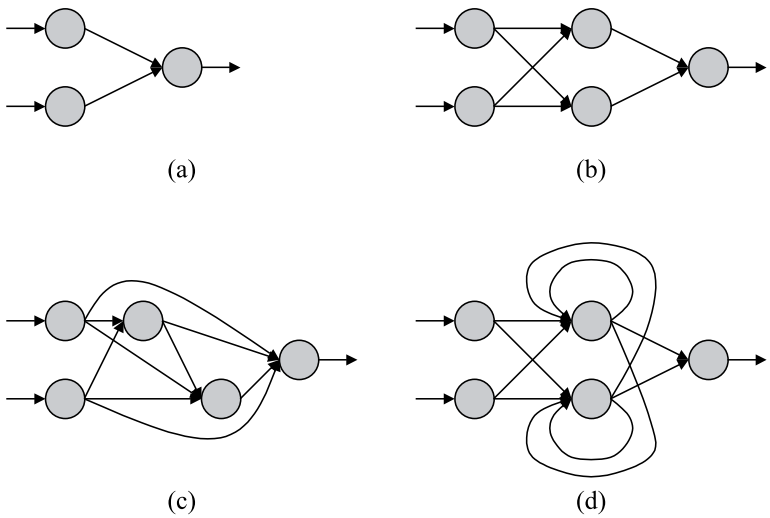

Figure 2.1: Examples of four different types of neural networks: (a) a perceptron, (b) a two-layer feed-forward network, (c) a general feed-forward network, and (d) a recurrent (Elman) network.

of interconnected nodes, or 'neurons'. Each neuron can receive input signals from other neurons via its incoming connections, and can send an output signal to other neurons over its outgoing connections. Neurons in the so-called 'input layer' receive signals from outside the network. Neurons in the so-called 'output layer' provide a reaction to the received signals over their outgoing connections. Neurons that are neither in the input layer nor in the output layer are called 'hidden neurons'.

For neuron n the output signal o_n is calculated as follows.

$$o_n = f((\sum_i w_i a_i) + b) \tag{2.1}$$

In this equation, w_i is a weight value attached to incoming connection i, a_i is the signal received via incoming connection i, b is a bias value, and f is a so-called 'activation function'. Two common activation functions are (i) a threshold function, that maps the output of the neuron to either 0 or 1, and (ii) a sigmoid function, that maps the output to a value in the range $[0, 1]$ (McCulloch and Pitts, 1943; Aleksander and Morton, 1990; Russell and Norvig, 2003).

Figure 2.1 displays examples of four common neural-network architectures, namely of (a) a perceptron, (b) a layered feed-forward network, (c) a general feed-forward network, and (d) a recurrent network.

A perceptron, of which an example is shown in Figure 2.1(a), is the simplest form of neural network (Rosenblatt, 1958; Minsky and Papert, 1988; Russell and Norvig, 2003). It contains only an input and an output layer.

A layered feed-forward network, of which an example is shown in Figure 2.1(b), contains hidden neurons organised in a sequence of layers. Each layer can receive input signals from the immediately-preceding layer only. A layered feed-forward network with one hidden layer is commonly called a 'two-layer feed-forward network' (the second layer being the output layer; by convention the input layer is not counted). A single-layer feed-forward network is a perceptron (Aleksander and Morton, 1990; Russell and Norvig, 2003).

A general feed-forward network, of which an example is shown in Figure 2.1(c), contains hidden neurons organised in a sequence. Each neuron can receive input signals from all neurons in the input layer, and from all neurons that are before it in the sequence. In other words, all possible feed-forward connections are allowed (Bishop, 1995).[1]

A feed-forward network is represented by an acyclic graph. A recurrent network is represented by a cyclic graph. It does not limit its connections to a feed-forward structure. A well-known form of recurrent network is the so-called 'Elman network', of which an example is shown in Figure 2.1(d) (Elman, 1990). An Elman network organises hidden neurons in layers. Recurrent connections are allowed between neurons within a layer. The recurrent connections are used to feed the output of neurons back into the network with a time-delay. Hence, they allow the network to support a short-term memory.

A neural network must be trained to emulate a desired function. This is commonly done with the help of a set of typical training samples, called the 'training set'. A well-known algorithm that trains a neural network is 'backpropagation'. This algorithm tests inputs from the training set, and propagates the error between the achieved and desired outputs back into the network, updating the connection weights (Aleksander and Morton, 1990; Russell and Norvig, 2003). When the average error on the training set is minimised, the network is validated using a 'test set' of typical samples, different from the training set. If the network achieves inferior results on the test set, this is usually caused by the network overfitting the training set. Common causes for overfitting are the use of a network with too many nodes, or the use of a training set with too few or untypical samples.

Neural networks are used in Chapters 3 and 4.

2.1.3 Evolutionary Artificial Neural Networks

Evolutionary artificial neural networks use the power of evolutionary algorithms to design neural networks. A typical application of evolutionary algorithms to neural-network design is an alternative for neural-network-training algorithms to determine the connection weights of the network. Other possibilities are the design of a network architecture and the tuning of network parameters. Combinations of these

[1] The most appropriate name for a general feed-forward network is 'feed-forward network'. In the literature, however, such networks are not conventional (Hertz, Krogh, and Palmer, 1991; Russell and Norvig, 2003), and the term 'feed-forward network' is often used to denote layered feed-forward networks. To avoid confusion I will use the term 'general feed-forward network' to denote networks that allow any feed-forward connection.

possibilities, such as designing the network architecture in parallel with determining the weight values, are also an option (Schaffer, Whitley, and Eschelman, 1992; Yao, 1995). A common design for an evolutionary algorithm that builds neural networks is as follows (cf. Albrecht, Reeves, and Steel, 1993; Yao, 1995).

- The neural networks are encoded as a chromosome by storing all connection weight values. If the network architecture is evolved in parallel with the weight determination, for each possible connection the chromosome also holds a bit that indicates whether the connection is present or absent.
- The fitness is defined by the error on a training set, where the fitness increases as the error decreases.
- Besides 'regular' genetic operators, often genetic operators are used that are tailored for neural-network evolution. Three examples of such genetic operators are (i) operators that switch neurons between networks, (ii) operators that enable or disable network connections, and (iii) operators that mutate neurons (Montana and Davis, 1989).

A problem that arises with neural network evolution is that structurally different networks may represent the same function. This is the problem of 'competing conventions' (Schaffer *et al.*, 1992).[2] Competing conventions increase the size of the solution space drastically, and marginalise the effect of crossover operators. While many solutions for competing conventions have been proposed (Hancock, 1992; Karunanithi, Das, and Whitley, 1992; Alba, Aldana, and Troya, 1993; Braun and Weisbrod, 1993; Thierens, Suykens, Vandewalle, and De Moor, 1993), some researchers consciously ignore the problem (Hancock, 1992), or restrict themselves to using only mutation operators ('genetic hill-climbing') or small populations (Schaffer *et al.*, 1992).

The four main advantages of using evolutionary algorithms to design neural networks instead of conventional training algorithms such as backpropagation are the following.

- Evolutionary algorithms can design the neural-network architecture in parallel with the weight determination, while conventional algorithms usually are restricted to just determining the weights.
- Evolutionary algorithms are designed to escape from local optima.
- Evolutionary algorithms only require a fitness function, while conventional algorithms often need more information (e.g., backpropagation needs the derivative of the error function).
- Evolutionary algorithms can design a neural network with *any* architecture, while conventional training algorithms are restricted to specific architectures (e.g., backpropagation is restricted to feed-forward networks).

[2] Alternative terms found in the literature are the 'permutation problem', the 'problem of isomorphism' and the 'structural/functional mapping problem'.

A disadvantage is that evolutionary algorithms are not suited for local optimisation. This means that when a solution close to the optimum is found, the evolutionary algorithm will, in general, not be able to seek out the actual optimum. The disadvantage can be resolved by applying a local-optimisation procedure (for example, one of the regular training algorithms) when it is observed that the evolutionary algorithm is unable to improve upon the best solution found.

Evolutionary artificial neural networks are used in Chapters 3 and 4.

2.1.4 Evolutionary Control

A 'plant' is a process that has input, output, and possibly an internal state. 'Plant control' aims at generating desired plant output by manipulating the input. 'Evolutionary control' uses evolutionary algorithms to design plant controllers. Although control engineers rarely use evolutionary techniques, they have been researched widely (Man and Tang, 1997; Fleming and Purhouse, 2001; Wang, Spronck, and Tracht, 2003). Evolutionary algorithms can be used to choose or tune parameters for controllers (e.g., the P(roportional), I(ntegral), and D(ifferential) values for PID-controllers), or to design new controllers from scratch. Evolutionary artificial neural networks can be used as controllers, and in that case are referred to as 'evolutionary neural controllers'.

Two complicating factors with plant control are that (i) the output need not react immediately to the input, and (ii) the internal state may cause the plant to behave differently in situations that, from the outside, seem to be equal. These complicating factors make it difficult, if not impossible, to determine whether an output of a plant is desirable. For plant control a training set, that couples desirable output values to input values, is therefore hard to design. Evolutionary control commonly analyses the behaviour of the controller over a test-run to determine the fitness.

The general design of an evolutionary-control experiment is illustrated in Figure 2.2. The experiment searches for a successful controller for a plant. The potential controller solutions are stored as chromosomes in a *population*. An *evolutionary algorithm selects* parent chromosomes from the population. It applies genetic operators to these parent chromosomes to generate new controllers. A newly generated controller is *tested* by placing it in a 'control loop'. In the control loop, the *controller* sends *control* signals to a *plant*, and receives *feedback* from the plant. The *test results* (indicating how successful the controller was in controlling the plant) are used by the evolutionary algorithm to assign a fitness value to the new controller. The evolutionary algorithm then *replaces* one of the chromosomes in the population with a chromosome that represents the new controller.

ELEGANCE, which is an acronym for Engineering Laboratory for Experiments with Genetic Algorithms for Neural Controller Evolution, is an environment I designed to do experiments with evolutionary neural controllers (Spronck, 1996; Spronck and Kerckhoffs, 1997). It is easily extendable and supports both feed-forward and recurrent neural controllers, a wide range of genetic operators and evolutionary algorithm parameters, and many different plants.[3]

[3] ELEGANCE is freely available through the Internet from the author's homepage.

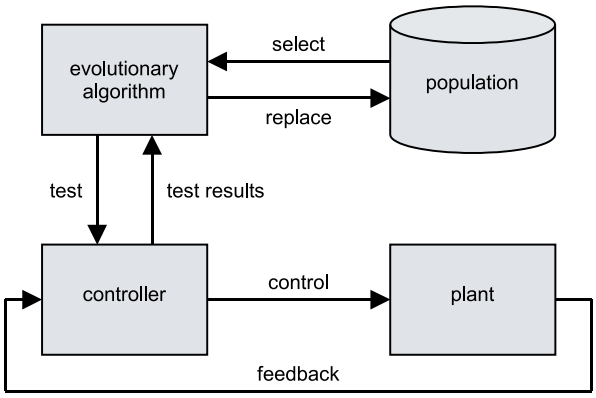

Figure 2.2: General design of an evolutionary control experiment.

Inspired by the encoding of Maniezzo (1993), the evolutionary algorithm employed in ELEGANCE allows evolving the network's weights in parallel with its architecture. The network is directly encoded into a chromosome consisting of an array of 'connection genes'. Each connection gene represents a single possible connection of the network and consists of a single bit and a real number. The bit represents the presence or absence of a connection and the real value specifies the weight of the connection. In this encoding scheme, even absent connections have a weight associated with them. The weight values of inactivated connections function as a kind of latent memory that can be reactivated by a mutation of the connection bit.

Evolutionary control is employed in Chapters 3, 4, and 6. ELEGANCE is used for experiments described in Chapters 3 and 4.

2.1.5 Reinforcement Learning

Reinforcement learning is used to train an agent to exhibit specific behaviour by rewarding and penalising agent actions coupled to states. State/action-pairs that drive the agent to desirable states are strengthened, while state/action-pairs that drive the agent to undesirable states are penalised. Rewards and penalties are usually awarded with a delay, because, when an agent has arrived at a state where a reward or penalty is given, not only the last action which the agent performed should receive the award, but the whole sequence of actions responsible for reaching the state (Mitchell, 1997; Sutton and Barto, 1998; Russell and Norvig, 2003).

Temporal-Difference (TD) learning is a form of reinforcement learning that learns a Q-function, which is an evaluation function for actions. Once a good Q-function has been derived, the success of new actions can be predicted and so the action with the highest expected reward in a given situation can be selected. A drawback of using TD-learning is that in practice many thousands of training iterations are required for the Q-function to converge (Mitchell, 1997). An example of the application of reinforcement learning in games, is TD-GAMMON, a program that learned to

play BACKGAMMON with TD-learning, using millions of training samples (Tesauro, 1992; Mitchell, 1997; Tesauro, 2002).

Reinforcement learning is similar to evolutionary control in the sense that both use an evaluation of the behaviour of an agent (or controller) to assign rewards and penalties. The major difference is that reinforcement learning is a gradient-search mechanism, that improves one solution by continuously making small changes to it, while evolutionary control examines each solution once and generates new solutions using undirected genetic operators.

Reinforcement learning is employed in Chapters 5 and 6.

2.2 Games

This section provides a concise overview of computer games. It presents a short history of games (2.2.1), an overview of different types of games (2.2.2), and the state of the art in game-AI research (2.2.3).

2.2.1 History

The very first game in the long lineage of commercial computer games was TENNIS FOR TWO, which is similar to PONG. It was created in 1958 by W. A. Higinbotham, and ran on a Brookhaven National Laboratory oscilloscope.[4] The first game that ran on a computer was SPACEWAR, created in 1962 by Steve Russell at MIT on a PDP-1 computer. In the game, illustrated in Figure 2.3, two players control spaceships that fire rockets at each other until one of them is destroyed (Levy, 1984). A version of SPACEWAR, named COMPUTER SPACE, was released by Magnavox as the first commercial console game in 1971. Magnavox' example was soon followed by other manufacturers who released game consoles, the most famous probably being the 1977 Atari VCS (Baratz, 2001).

Inexpensive micro-computers have been sold since the early 1970s. They became popular in 1977 with the release of the TRS-80 and the Apple II computers. These computers were meant both for both business and home users. For the latter group, games were built and published by dedicated game companies such as Electronic Arts, Infocom, Origin, Sierra, and SSI. While originally game developers needed to support a wide variety of computers, in the mid-1980s the IBM-PC became the industry standard for home computing and thus for home gaming. In parallel development, gaming consoles (dedicated game computers that are hooked up to a television set) became popular, starting with the Nintendo Entertainment System in 1986 (Baratz, 2001).

[4] Many argue that the very first game was TIC-TAC-TOE, programmed in 1952 by A. S. Douglas for the EDSAC computer, which used a cathode-ray tube to display the playing grid. However, in my opinion TIC-TAC-TOE is an analytical game, and as such does not deserve the title of first *commercial* computer game. Note that computers played analytical games even before 1952: in 1951 D. G. Prinz built a CHESS-playing program, that was the first program to solve a CHESS problem (Van den Herik, 1983).

Figure 2.3: SPACEWAR, great-great-grand-parent of modern games.

The continuous advances in processing power and capabilities of home computers, caused games to become increasingly complex. While in the 1980s a team of five people could create a top-rated game, in the 1990s game-development teams consisted of hundreds of people. The cost of producing a game grew accordingly.

Since the start of the twenty-first century, the game industry has grown to surpass the multi-billion-dollar Hollywood movie industry in revenues (Fairclough *et al.*, 2001; Snider, 2002). The market for PC and console games now only allows for large game-development companies, supported by wealthy publishers. For the smaller developers, a new market has opened up with handheld gaming. It is, however, only a matter of time before the domain of handheld game development also is taken over by large game developers (Spronck and Van den Herik, 2003).

For a long time the processing power of computers was mainly invested into creating better graphics. In the late 1990s specialised 3D video cards became affordable and widespread. This freed up processing power for other game-play features, such as artificial intelligence (Tozour, 2002b). Game-AI programming has become an important activity in game development, instead of something that is added in the last weeks before a game is released. Therefore the subject of this thesis, game AI, is relevant for the game industry as it exists today.

2.2.2 Game Types

Games can be divided into different categories. There is no general consensus on what those categories are.[5] My view is that there are six categories of games: action games, adventure games, puzzles, role-playing games, simulations, and strategy games. I discuss the different categories below.

Action: Action games are games that require players to use mainly their reflexes to beat the game. The five main types of action games are *arcade games* (such as

[5]For example, Fairclough *et al.* (2001) distinguish 'action games', 'adventure games', 'role-playing games' and 'strategy games'. Schaeffer (2001) adds to these 'god games' and 'sports games'. Laird and Van Lent (2001) have a similar view, but make a clear distinction between 'team sports games' and 'individual sports games'.

PACMAN), *platform games* (such as PRINCE OF PERSIA), *sports games* (such as FIFA SOCCER), *3D shooters* (such as QUAKE), and *3D sneakers* (such as THIEF). Nowadays the first two types have almost died out, while the others are arguably the most popular types of games available. The game AI in action games controls individual agents on an operational and tactical level.

Adventure: Adventure games are story-driven games that require players to follow a specific path towards the end of the game. The path is littered with puzzles of all kinds that players must solve, using their intellectual skills. The two main types of adventure games are *text adventures* or *interactive fiction* (such as ZORK), and *graphical adventures* (such as KING'S QUEST). Nowadays the adventure-game genre seems to have almost died out, although amateurs, some surprisingly talented, still produce these games (Montfort, 2004). Characters in adventure games can only react in a pre-defined way to specific player actions. As such, game AI is absent for adventure games. [6]

Puzzle: Puzzle games are games that require players to apply their intellectual skills to solving a puzzle. The two main types of puzzle games are *time-free puzzles* (such as SOKOBAN), and *time-constrained puzzles* (such as TETRIS). Puzzle games are, in general, not very popular, except for handheld computers. Puzzles do not require game AI.

Role-playing: Computer role-playing games (CRPGs) are story-driven games that require players to assume the role of a game character. Players are sent on a quest, usually with a fantasy or a science-fiction theme. The quest mainly involves exploration and tactical combat. The two main types of CRPGs are *single-player CRPGs* (such as BALDUR'S GATE), and *massive multiplayer online games* (such as EVERQUEST). After almost having died out in the 1990s, CRPGs have become quite popular again nowadays. The game AI in CRPGs controls individual agents on an operational and tactical level.

Simulation: Simulation games are games that require players to observe and interact with a simulation. The two main types of simulation games are *god games* (such as THE SIMS), and *vehicle simulations* (such as FLIGHT SIMULATOR). Simulations always have been fairly popular. The amount of game AI that pervades a simulation game depends on the level of realism of the simulation.

Strategy: Strategy games are games that require players to use their strategic and tactical skills to guide a group of agents to victory. The two main types of strategy games are *turn-based strategy games* (such as CIVILIZATION and RAILROAD TYCOON), and *real-time strategy games* (such as WARCRAFT). Strategy games have been popular since the 1990s. The game AI in strategy games controls large groups of agents on an operational, tactical and strategic level.

[6] Some adventure games, especially text adventures, contain characters that exhibit seemingly intelligent behaviour, but in general their choice of actions is based on simple probability. They are not in the game as opponents for the player, but as puzzles to be solved (Lebling, 1980).

Type	Games	Sections
Action	QUAKE	4.2
Role-playing	Simulated CRPG	5.2–5.4
Role-playing	NEVERWINTER NIGHTS	5.5
Strategy	Duelling spaceships	4.1
Strategy	WARGUS	6.2–6.4

Table 2.1: Game types investigated in the thesis.

Many games that are in existence today fall into more than one of the categories. To stand out, game developers attempt to combine game genres to create an original game that exhibits the best of different categories (Slater, 2002). For instance, vehicle simulations are often enhanced with action elements, and action games are often enhanced with elements from strategy games. Complex game AI is encountered mainly in role-playing games and strategy games.

Table 2.1 lists the game types discussed in this thesis. From left to right, the three columns represent (i) the game type, (ii) the games of this type discussed, and (iii) the corresponding thesis sections.

2.2.3 Game-AI Research

Game AI is of interest to two different groups, namely (i) game developers, who aspire to have game AI keep up with game enhancements, and (ii) academic researchers, who profess to have a high-level view of the field of game AI. Surprisingly, there is little communication between these two groups (Sawyer, 2002). Game developers complain that academics fail to get out of their ivory tower to help them with the *practical* implementation of game AI (Laird, 2000; Tozour, 2002b). Academics claim they cannot get their foot in the door of game development, because of industry secrets (Sawyer, 2002; Buro, 2003a), tight schedules (Sawyer, 2002), and lack of funding (Laird and Van Lent, 2001; Sawyer, 2002). Consequently, game developers and game researchers tend to remain in their own communities.

Fortunately, this trend is changing. Game developers recognise they need help from academic communities to implement game AI that can cope with the complexities of modern games (Laird and Van Lent, 2001; Sawyer, 2002; Rabin, 2004b). Game resources are freed up for more advanced game AI (Laird, 2000). Academics are allowed access to modern game engines for their research (Laird, 2000), through open source, or through toolsets released with the games. Nowadays, many academic AI researchers attend game development conferences, and occasionally a game developer visits an academic conference on game-AI research.

Not only game developers can benefit from the work of AI researchers, but AI researchers have much to gain from the work of game developers as well. Since the goal of game AI is to make human players believe that their opponents are actually controlled by other humans (Laird and Van Lent, 2001; Sawyer, 2002; Livingstone and McGlinchey, 2004), modern games are nothing less than a practical implementa-

tion of a Turing Test (Turing, 1950). Even small steps that AI researchers can take towards human-like game AI are welcomed by game developers, and, when implemented in an actual game, will be tested out in practice (Laird and Van Lent, 2001). Furthermore, games are a popular pastime, which may help to attract students to the field of AI research, and gain attention from popular media.

This thesis aims at bridging the gap between academic research and the daily practice of game development. It investigates the application of machine-learning techniques to game AI. A major requirement of the techniques investigated is their practical applicability in modern games.

2.3 Machine Learning and Game AI

This section clarifies the three different ways in which machine learning can be applied to game AI, namely offline learning (2.3.1), supervised learning (2.3.2), and online learning (2.3.3). It also discusses the requirements that online learning of game AI must meet (2.3.4).

2.3.1 Offline Learning

'Offline learning' of game AI is learning that takes place while the game is not being played by a human (Charles and McGlinchey, 2004; Funge, 2004). This can be learning from samples or learning by self-play (i.e., the computer controlling all sides in the game). A typical application of offline learning is tuning game-AI parameters during the 'quality assurance' phase of game development. A more advanced application is creating new tactics for opponents by self-play.

Although offline learning is a common technique used in analytical games (Tesauro, 1992; Schaeffer, 1997; Schaeffer, Billings, Peña, and Szafron, 1999; Donkers, 2003; Enzenberger, 2003; Kocsis, 2003; Van der Werf, 2004; Winands, 2004) and is sporadically used in academic research of commercial games (Ballard, 1997; Laramée, 2002a; McGlinchey, 2003; Spronck and Van den Herik, 2003), the literature provides little or no examples of offline learning used by professional game developers, other than tweaking a few parameters (Biasillo, 2002; Woodcock, 2002). Neither did my own contacts with game developers turn up any evidence of offline learning in professional games. This is somewhat surprising, since offline learning takes place entirely 'in-house', and therefore is the least risky application of machine learning to games. Chan *et al.* (2004) surmise that the use of offline learning of game AI to help game designers and programmers for the purpose of quality assurance is the first step to introduce machine-learning techniques in the game industry.

In this thesis offline learning in games is discussed in Chapters 3, 4, and 6.

2.3.2 Supervised Learning

'Supervised learning' of game AI takes places while the game is being played by a human. It implements changes to the game AI by processing immediate feedback on

any decision that the game AI makes. The feedback indicates whether a decision is desired or undesired. With supervised learning of game AI the human player controls what is being learned, either by providing the game AI with samples of behaviour to be imitated, or by rewarding desired behaviour and penalising undesired behaviour.

When supervised learning is part of a game, it requires the cooperation of the human player, i.e., the learning is part of the game-play design. Very few games incorporate supervised learning. Two well-known examples of such games are CREATURES and BLACK & WHITE. In both games, the agent behaviour is partly determined by a learning structure (the agent's 'brain'). In CREATURES the learning structure consists of a neural network (Adamatzky, 2000), and in BLACK & WHITE it consists of a decision tree and perceptrons (Evans, 2001 & 2002; Fu and Houlette, 2004). The human player trains the learning structure by rewarding agents when they exhibit desired behaviour, and penalising them when they exhibit undesired behaviour.

This thesis is on automatic learning of game AI. Supervised learning is not automatic, for it requires human intervention. Therefore, supervised learning will not be discussed further in this thesis.

2.3.3 Online Learning

'Online learning' of game AI is learning that takes place while the game is being played by a human (Charles and McGlinchey, 2004; Funge, 2004).[7] Through online learning, game AI automatically adapts in accordance with the human player's style and tactics. There are two main reasons to implement adaptive game AI, namely (i) the game AI makes exploitable mistakes, which makes the game too easy, and (ii) the game AI's skill is not in the same league as the human player's skill, which makes the game either too easy or too hard. Both reasons, if neglected, are detrimental to a game's entertainment value.

Some academic research has investigated online learning in games (Demasi and Cruz, 2002; Laramée, 2002b; Mommersteeg, 2002; Demasi and Cruz, 2003; Aha and Molineaux, 2004; Graepel *et al.*, 2004; Le Hy, Arrigoni, Bessière, and Lebeltel, 2004; Jones and Goel, 2004; Leen and Fyfe, 2004; Spronck, Sprinkhuizen-Kuyper, and Postma, 2004c; Ulam, Goel, and Jones, 2004). In practice, however, game publishers are reluctant to release games with online-learning capabilities (Funge, 2004). Their main fear is that the game learns inferior behaviour (Woodcock, 2002; Charles and Livingstone, 2004). Therefore, the few games that contain online learning, only do so in a severely limited sense, in order to run as little risk as possible (Charles and Livingstone, 2004).

Two less-risky possibilities for online learning in games are (i) to change automatically a few parameters (e.g., in NASCAR RACING 2003 SEASON and THE FALL OF MAX PAYNE), and (ii) to switch automatically between several manually-designed

[7] Supervised learning (2.3.2) also takes place online. Therefore, to be absolutely clear, 'online learning' should be named 'unsupervised online learning'. However, in the literature, when learning is mentioned, it is usually assumed that unsupervised learning is meant. This thesis does not investigate supervised learning. I therefore use the shorter term 'online learning' to refer to 'unsupervised online learning'.

varieties of the game AI, such as different formations of enemy groups (e.g., in DESCENT 3: MERCENARY and WWII: FRONTLINE COMMAND). While these simple attempts to implement adaptive game AI can be surprisingly effective (Funge, 2004), they are not always appreciated by game players.[8]

In this thesis online learning in games is discussed in Chapters 4, 5, and 6.[9]

2.3.4 Online Learning Requirements

After a search through the literature, personal communication with game developers, and applying our own insights to the subject matter, we arrived at a list of four computational and four functional requirements, which online adaptive game AI must meet to be applicable in practice.

The computational requirements are necessities: failure of an online adaptive-game-AI technique to meet the computational requirements makes it useless in practice. The functional requirements are not so much necessities, as strong preferences by game developers: failure of an online adaptive-game-AI technique to meet the functional requirements means that game developers will be unwilling to include it in their games, even when it yields good results and meets all four computational requirements. The four computational requirements are the following.

Speed: Online learning in games must be computationally fast, since learning takes place during game-play (Laird and Van Lent, 2001; Nareyek, 2002; Charles and Livingstone, 2004; Funge, 2004).

Effectiveness: Online learning in games must create effective game AI during the whole learning process, to avoid it becoming inferior to manually-designed game AI, thus diminishing the entertainment value for the human player (Charles and Livingstone, 2004; Funge, 2004).[10]

Robustness: Online learning in games has to be robust with respect to the randomness inherent in most games (Chan *et al.*, 2004; Funge, 2004).

Efficiency: Online learning in games must be efficient with respect to the number of trials needed to achieve successful game AI, since in a single game, a player experiences only a limited number of encounters with similar groups of opponents.

[8] For instance, after the release of THE FALL OF MAX PAYNE, many players complained that if they played the game too well, the opponents soon achieved capabilities that made them almost impossible to defeat. Players started to take deliberate damage, in order to fool the game into assuming the difficulty level should not be increased.

[9] Note that the term 'online' as used in this thesis should not be confused with the popular meaning of 'online' to refer to activities that are performed over the internet. For instance, the work of Baxter, Tridgell, and Waever (1998) in which reinforcement learning is applied to improve a CHESS evaluation function using games played through the internet, is actually an example of offline learning, since the evaluation function is changed only after the games have been played.

[10] Usually, the occasional occurrence of a non-challenging agent is permissible, since the player will attribute an occasional easy win to luck. Note that, if adaptive game AI meets this requirement, the main fear of game publishers, that agents will learn inferior behaviour, is resolved.

The four functional requirements are the following.[11]

Clarity: Online learning in games must produce easily interpretable results, because game developers distrust learning techniques of which the results are hard to understand.

Variety: Online learning in games must produce a variety of different behaviours, because agents that exhibit predictable behaviour are less entertaining than agents that exhibit unpredictable behaviour.

Consistency: The average number of trials needed for adaptive game AI to produce successful results should have a high consistency, i.e., a low variance, to ensure that it is rare for players to find that learning in a game takes exceptionally long.

Scalability: Online learning in games must be able to scale the difficulty level of its results to the experience level of the human player (Lidén, 2004).

To meet the four computational requirements, an online learning algorithm must be of 'high performance'. According to Michalewicz and Fogel (2000), the two main factors of importance when attempting to achieve high performance for a learning mechanism are the exclusion of randomness and the addition of domain-specific knowledge. Since randomness is inherent in most games, it cannot be excluded. Therefore, it is imperative that the learning process is based on domain-specific knowledge (Manslow, 2002).

Obviously, it is hard to create an online-learning technique for games that meets all the eight requirements. However, the 'dynamic scripting' technique, discussed in Chapter 5, is designed to do just that.

2.4 Chapter Summary

This chapter provided background information on the research in this thesis. It discussed machine-learning techniques used in the research (evolutionary algorithms, artificial neural networks, evolutionary artificial neural networks, evolutionary control, and reinforcement learning), and gave an overview of commercial computer games and game-AI research. It distinguished three different ways in which machine learning can be applied to game AI, namely (i) offline learning, (ii) supervised learning, and (iii) online learning. For online learning four computational requirements were listed, namely the requirements of (i) speed, (ii) effectiveness, (iii) robustness, and (iv) efficiency. Furthermore, four functional requirements were listed, namely the requirements of (i) clarity, (ii) variety, (iii) consistency, and (iv) scalability. The focus of this thesis is on unsupervised learning, that is, on offline and online learning.

[11] The first two functional requirements, the requirements of clarity and variety, were expressed by three of the lead developers of BioWare Corp, during a personal exchange I had with them in 2003.

Chapter 3

Doping in Agent Control

Better Living Through Chemistry.
— Advertising slogan of Monsanto Corporation.

Agents in games have a task to accomplish; usually, it is defeating a human player. Game AI controls the behaviour of the agents in game environments. The present chapter[1] investigates evolutionary control of agents in game-like environments. A game-like environment has two major characteristics with respect to agents, namely (i) agents have only a limited view of the environment, and (ii) agents can interact with the environment to accomplish their tasks.

Evolutionary control is an effective technique for creating the controllers of the agents (2.1.4). To achieve good results, evolutionary control must deal with the 'problem of hard instances'. This chapter explores a novel technique designed to alleviate the problem of hard instances, called the 'Doping-driven Evolutionary Control Algorithm' (DECA). Section 3.1 describes the problem of hard instances, and introduces DECA. Section 3.2 describes the experimental procedure employed for evaluating DECA. Sections 3.3 and 3.4 are devoted to two experiments that confirm DECA's effectiveness. Section 3.5 provides a general discussion of the experimental results. A summary of the chapter is provided in Section 3.6.

3.1 DECA and the Problem of Hard Instances

Agents in game-like environments have a task to accomplish. A 'task instance' is a specific example of the environment in which the agent resides. Evolutionary control can be used to determine the agent's behaviour in the environment (2.1.4). Evolutionary control tends to favour controllers that solve easy task instances, but that fail to solve the hard ones. This phenomenon is called 'the problem of hard instances' (Spronck, Sprinkhuizen-Kuyper, and Postma, 2001a). It can be alleviated by the Doping-driven Evolutionary Control Algorithm (DECA), which is based on

[1] This chapter is based on a paper by Spronck, Sprinkhuizen-Kuyper, Postma, and Kortmann (2003c), and a submitted paper by Spronck, Sprinkhuizen-Kuyper, and Postma (2005).

the notion of 'doping'. This section explains the problem of hard instances (3.1.1), provides background information on doping (3.1.2), and defines and explains DECA (3.1.3). From hereon I will refer to a 'task instance' with the shorter term 'instance'.

3.1.1 The Problem of Hard Instances

Evolutionary learning is effective for creating the controllers of situated agents (Arkin, 1998). When applying evolutionary learning to controller design, the mapping executed by the controller is generated by setting the controller parameters. The quality of controllers is defined in terms of an appropriate measure as determined by the fitness function. In general, the fitness function is based on the evaluation of a controller on a series of typical instances varying in difficulty from easy to hard. An easy instance is an instance for which a solution can be found easily, i.e., in the search space, solutions to easy instances are abundant and located in 'flat' regions of the search space. In contrast, a hard instance is an instance for which it is difficult to find a solution, i.e., in the search space, solutions to hard instances are rare and located at 'peaks' surrounded by inferior solutions (Spronck *et al.*, 2001a).

In the evolutionary learning process new controllers are generated by recombining elements of previously-generated controllers, favouring those that have a relatively high fitness. Obviously, a controller that solves at least one of the instances is assigned a higher fitness value than one that solves no instances at all. Since it is very likely that controllers that cope with easy instances are discovered before those that cope with harder instances, the performance on the easy instances determines the course of the evolutionary process to a great extent. Therefore, the evolutionary search is more or less confined to the regions of search space where most of the solutions to easy instances reside. Unless a good solution that covers both easy and hard instances is found in the vicinity of these regions, the end result is a controller that handles easy instances well, but fails on the hard ones. This is called 'the problem of hard instances'.

If the problem of hard instances is not dealt with, evolutionary algorithms are bound to produce inferior solutions to task control problems. To deal with the problem of hard instances, I propose the Doping-driven Evolutionary Control Algorithm (DECA). DECA is based on the notion of 'doping', which is explained below.

3.1.2 Doping

Doping is defined as the addition of some very good solutions to a population (usually the initial one) in order to facilitate the evolution process. These solutions may be generated by a different algorithm or may express the user's knowledge about the problem domain (Dumitrescu, Lazzerini, Jain, and Dumitrescu, 2000). Common terms used for similar techniques are 'seeding', 'case injection' (Louis, 2002) and 'infusion' (Spronck *et al.*, 2001a). If there are differences between the exact meanings of these terms, they are not well defined. The term 'seeding' is used in the literature most often. It refers to the injection of any kind of genetic material into a population.

I chose to use the term 'doping' to refer to the injection of complete solutions into a population, rather than the injection of any kind of genetic material.

The application of doping (or seeding) is restricted to those cases where it is important to retain specific genetic material in the population (Dumitrescu *et al.*, 2000). The best-known example is in the 'messy Genetic Algorithm' (mGA), where in the primordial phase of the evolution the population is doped with all possible building blocks of a specific length (Goldberg, Deb, and Korb, 1991). Sometimes doping takes the form of inserting manually-designed solutions into the initial population. An example is the work of Matthews *et al.* (2000) on a problem in land-use planning where the initial population was doped with heuristic and expert-based solutions. In Case-Initialised Genetic Algorithms (Louis and Johnson, 1999), a solution to a problem similar to the target problem is inserted in the initial population to facilitate the evolution process in finding a good solution to the target problem. Grefenstette and Ramsey (1992) created an initial population that consisted of 50 per cent solutions that worked well in the past, 25 per cent manually-designed solutions for the problem in general, and only 25 per cent solutions generated randomly.

While the examples mentioned above demonstrate beneficial effects of doping, it should be considered whether doping can be detrimental to the evolution process. Doping genetic material that is unrelated to any known solution, as is done in the mGA, does little harm to the final solution. However, doping an initial population with known solutions may lead to inferior results. The reason is that within a population of random solutions, a fairly good solution is likely to have the highest fitness, which leads to convergence to a local optimum in the vicinity of the doped solution. The evolution process is used as a local optimisation process, rather than as a method to scan the search space. Good solutions that are too remote from the doped solution are likely to be missed. In order for doping to yield good results in task-control problems, the evolutionary process needs to be biased to deal with hard instances. This is exactly what is done in DECA as will be detailed below.

3.1.3 DECA

The Doping-driven Evolutionary Control Algorithm (DECA) ensures that the evolutionary search is confined to those regions of the search space where the solutions to hard instances are likely to be found. In order to achieve the bias, DECA applies doping as described in the following six steps.

1. *Training-set design*: Select a series of instances that encompass most or all relevant characteristics of a task.
2. *Hard-instance selection*: Identify a hard instance that encompasses most of the relevant characteristics.
3. *Hard-instance evolution*: Evolve a good solution to the hard instance selected in the previous step.
4. *Initialisation*: Generate a random population and 'dope' this population with the solution evolved in the previous step.

5. *Evolution*: Evolve good solutions to the complete series of instances selected in step 1 using the doped population.

6. *Validation*: Evaluate the validity of the evolved solution on a new selection of instances.

If no domain knowledge is available to select hard instances in step 2, a (time-consuming but generally applicable) way to identify hard instances is to attempt to evolve separate solutions to all the instances in the training set, and observe for which instances the evolution process takes the longest on average.

DECA is expected to yield good results because I assume that there is an asymmetry in the search space with respect to easy and hard solutions (i.e., local minima of the fitness function). Solutions to easy instances are readily found in the vicinity of solutions to hard instances, whereas the reverse is not true. The asymmetry is caused by the abundance of solutions to easy instances and the relative scarcity of solutions to hard instances. The validity of this assumption is discussed in more detail in Subsection 3.5.1.

3.2 Experimental Procedure

To evaluate the effectiveness of DECA, two experiments were performed with two different tasks. The first task is a box-pushing task wherein a robot has to push a box between two walls. The second task is a food-gathering task in which an agent has to collect food while avoiding to be damaged. For both tasks neural controllers were used, which are suitable adaptive structures for situated agents (Arkin, 1998). The weights and architectures of the controllers were generated using an evolutionary algorithm, using the ELEGANCE environment (2.1.4).

Preliminary experiments with the evolution of a neural box-pushing controller indicated that a recurrent neural controller outperforms various kinds of feed-forward controllers on this particular task (Sprinkhuizen-Kuyper, Postma, and Kortmann, 2000b). I therefore decided for both experiments to use a neural network configuration that gave the best results in the preliminary experiments, namely an Elman network (2.1.2) with a maximum of four hidden nodes, and the network output values constrained by applying a sigmoid function.

In the experiments the following six genetic operators were employed, which were found to perform well in evolving solutions for other neural control problems (Spronck, 1996).

- *Uniform crossover*: Child chromosomes are created by copying each allele from one of two parents, each parent having a 50 per cent chance of being selected for each allele (Goldberg, 1989).

- *Biased weight mutation* (Montana and Davis, 1989): Child chromosomes are copies of parent chromosomes, with each weight having a 5 per cent chance to be mutated by adding a random value selected from the range $[-0.3, 0.3]$.

- *Biased nodes mutation* (Montana and Davis, 1989): Child chromosomes are copies of parent chromosomes, with all the input weights of one randomly selected node changed by adding a random value selected from the range $[-0.3, 0.3]$.
- *Nodes crossover* (Montana and Davis, 1989): Child chromosomes are created by copying each of their nodes (including their input connections) from one of two parents, each parent having a 50 per cent chance of being selected for each node.
- *Node existence mutation* (Spronck, 1996): Child chromosomes are copies of parent chromosomes, with a 95 per cent chance of having all incoming and outgoing connections of one randomly-selected node being removed, and a 5 per cent chance of having all absent connections of a randomly-selected node being activated.
- *Connectivity mutation* (Spronck, 1996): Child chromosomes are copies of parent chromosomes, with each connection having a probability of 5 per cent to switch from being connected to being disconnected and vice versa.

During evolution, one of these six operators was selected at random. For the crossover operators, I arbitrarily decided to add only the fittest of the two children to the population, while the other child was rejected. To alleviate the problem of competing conventions (2.1.3) the hidden nodes of the parents were rearranged to make their signs match (insofar as possible) before a crossover took place (Thierens *et al.*, 1993). Newly-generated individuals replaced existing individuals in the population, while taking into account elitism. Crowding (Goldberg, 1989) with a factor of 3 was used as replacement policy. For the selection process, size k tournament selection (Goldberg and Deb, 1991) was used, with $k = 2$ for the box-pushing experiment and $k = 3$ for the food-gathering experiment.

In all experiments, the population size was equal to 100 and real-valued weights were used. In preliminary experiments larger population sizes were tested, with a maximum of 250, but these did not give significantly better results. Based on the observed convergence rates, I set the maximum number of generations to 35 for the box-pushing experiment, and to 30 for the food-gathering experiment. Preliminary experiments showed that in rare cases slightly better solutions could be achieved if the evolution was allowed to continue for more generations, but in my view the considerable increase in computation time required was not worth the small improvement in performance.

Having discussed the experimental procedure, I now turn to the description of the two experiments to evaluate the effectiveness of DECA.

3.3 Box-Pushing Behaviour

The box-pushing task is the first task to evaluate DECA. The task involves the pushing of a box between two walls. A simpler version of the task was introduced

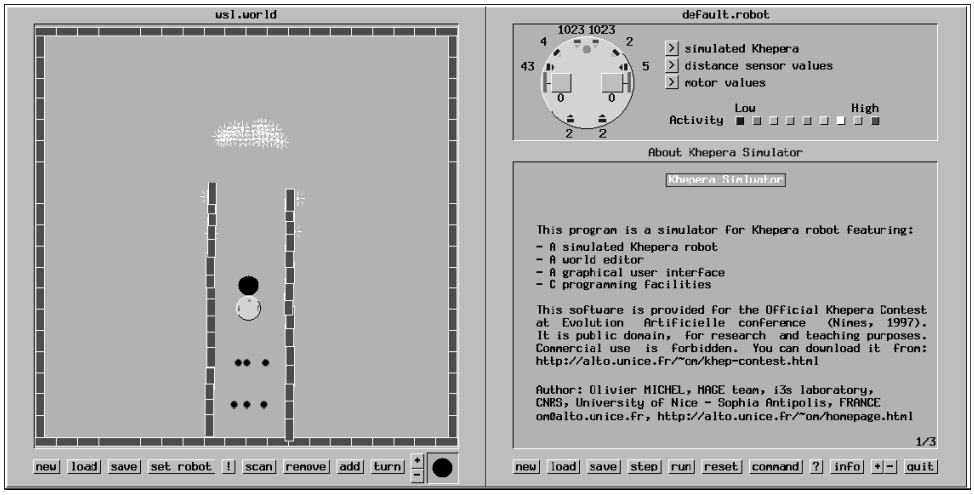

Figure 3.1: Simulation environment of the Khepera robot.

by Lee, Hallam, and Lund (1997). Pushing an object (in this case a circular box) between two walls is an elementary behaviour that is relevant in, for instance, the game of robot soccer in which a ball has to be pushed towards the opponent's goal (Asada and Kitano, 1999). The task is non-trivial, because it requires the agent to adapt continuously to the position of the ball as perceived through the noisy sensors. Elementary behaviours, of which the box-pushing task is only an example, are believed to underlie more complex behaviours such as target following, navigation and object manipulation. I describe the box-pushing task in Subsection 3.3.1, present the achieved results using DECA in Subsection 3.3.2, and provide a discussion of the results in Subsection 3.3.3.

3.3.1 The Box-Pushing Task

To study box-pushing behaviour, a publicly available mobile robot simulator was employed. The simulator is based on the widely used mobile robot Khepera (Mondada, Franzi, and Jenne, 1993). It is illustrated in Figure 3.1. The square area on the left side is the robot world and measures 1000×1000 units. The grey circle represents the robot, the black circle the box, and the six small black dots the starting positions of the box (the upper three dots) and the robot (the lower three dots). The starting positions can be combined to nine instances, that differ in the initial configuration of robot and box (illustrated in Figure 3.4).

The (simulated) Khepera displayed in Figure 3.2 is equipped with eight sensors and two motors, one for each of the wheels. The sensors provide the robot with proximity values. For the purpose of the experiment, the simulator was coupled to the ELEGANCE environment. The Khepera simulation is controlled by a neural network

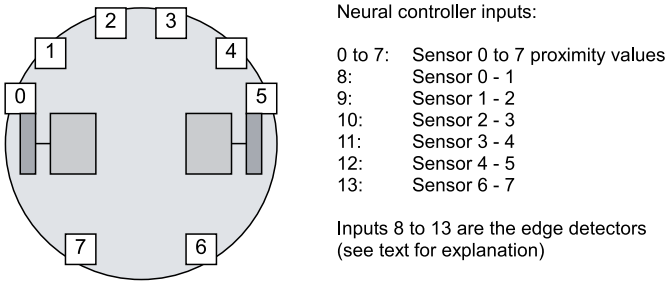

Figure 3.2: Schematic overview of the Khepera robot, with a mapping to the neural controller inputs.

with fourteen inputs, provided by the eight proximity sensors and six additional virtual 'edge-detector' sensors. The outputs of the virtual sensors are defined as the differences in proximity values between all pairs of neighbouring sensors, e.g., sensor 8 gets the proximity value of sensor 0, from which the proximity value of sensor 1 is subtracted. It is important to note that the Khepera simulation is stochastic because the sensors and controller outputs generate noisy signals.

The motors driving the wheels are controlled by the outputs of two neural networks, one for the left and one for the right wheel. Exploiting the mirror symmetry of the perception-to-action mapping, the two neural networks are identical except for the mapping of sensors to network inputs and the definition of the signs of the edge-detecting inputs. Figure 3.3 illustrates the different mapping and signs for both networks. In the figure, the small rectangles at the left of the neural networks indicate the sensors. In these rectangles, $x - y$ indicates an edge detector in which the value of sensor y is subtracted from the value of sensor x.

The task set to the simulated robot was to push the box as far away as possible from its starting position within a limited period of time. Figure 3.4 illustrates the nine instances numbered 0 to 8. The box-pushing task is difficult because the robot (i) must identify the box, (ii) must remain behind the box while pushing, (iii) must prevent the box from getting stuck, and (iv) must deal with noise generated by the sensors and the motor controls. Preliminary experiments revealed that the nine instances exhibited these difficulties in various degrees. For instance, in instances 0, 4, and 8, the box is positioned directly in front of the robot, which means the robot can perform its task by simply moving forward and correcting for small deviations. Instances 2 and 6 are harder since the initial separation of the robot and box is larger than in instances 0, 4, and 8. Instances 3 and 5 can be considered the most difficult because in these tasks the robot suffers more from the roughness of the walls than in any of the other instances (Spronck *et al.*, 2001a).

At first glance it may seem that instances 2 and 6 are equally difficult, if not more difficult than instances 3 and 5. However, I found that, in general, evolving a controller for instances 3 and 5 takes considerably longer than for instances 2 and

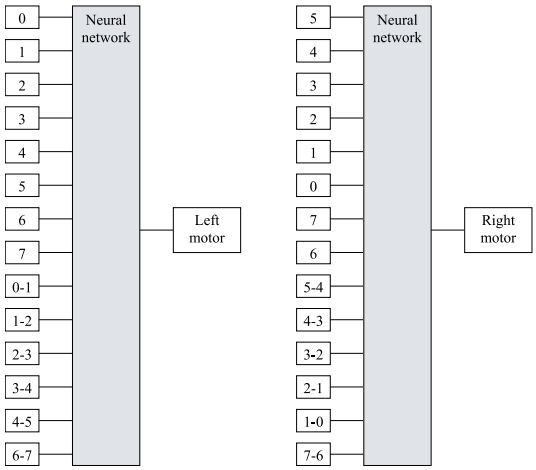

Figure 3.3: The two almost identical networks that drive the left and right robot motors. The network inputs are proximity values derived from the robot in Figure 3.2.

6, with worse results for the final fitness values reached. The explanation for these counterintuitive results is as follows. In instances 2 and 6, the robot travels a longer distance from its starting position to the box than in instances 3 and 5. The longer distance allows the robot more time and more room to manoeuvre to a good position to slide the box along the wall. In instances 2 and 6 the robot learns to position itself directly 'below' the centre of the box. In instances 3 and 5, the robot has less time and less room to manoeuvre to a good position, and so it tends to push the box 'sideways', thereby hitting the wall under an inconvenient angle. This is illustrated in Figure 3.4. In this figure, the circles shown are the robot (largest circles) and the circular box (slightly smaller circles) at their initial (bottom) and final (top) positions. The lines connecting the initial to the final positions represent typical paths followed by the robot and the box.

Sprinkhuizen-Kuyper, Kortmann, and Postma (2000a) determined a suitable fitness function to measure the success of the robot's behaviour in this experimental setup. I copied their fitness function, which is defined as follows. If $robot_t$ is the position of the robot at time t, and box_t is the position of the box at time t, the fitness value assigned to a robot upon completion of a single instance i is defined as follows.

$$F_i = d_i(box_T, box_0) - \frac{1}{2}\, d_i(box_T, robot_T) \tag{3.1}$$

In this equation, $d_i(box_T, box_0)$ represents the Euclidian distance between the initial ($t = 0$) and final positions ($t = T$) of the box, and $d_i(box_T, robot_T)$ the Euclidian distance between the robot and the box at their final positions for instance i (all distances are calculated between the centres of the objects). An experimental trial comprises $T = 100$ steps on each of the nine instances. The average fitness F_{avg} on

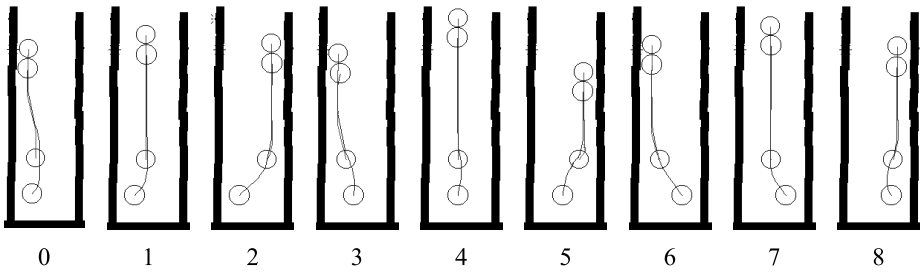

Figure 3.4: The nine instances (0 to 8) and typical trajectories of the robot and the box. Note that the roughness of the walls hinders the robot in sliding the box along a wall.

a trial is defined as the average fitness over all instances, i.e., $F_{avg} = \frac{1}{9}\sum_{i=0}^{8} F_i$.

In the present experiment, to reduce the effect of the noise the overall fitness F was defined as the average of the trial fitness values over a number of R repetitions of trials, i.e., $F = \frac{1}{R}\sum_{r=1}^{R} F_{avg}^r$, with F_{avg}^r representing the average fitness F_{avg} obtained at the r–th repetition. Computational resources constrained the number of repetitions. The number of repetitions was varied between $R = 1$ and $R = 100$ depending on the following considerations. In preliminary experiments I already determined that controllers with a fitness value of 250 or less on a single trial are inferior, and remain inferior on replications of the trial.[2] The contribution of inferior controllers to the evolution process is limited, and consequently their ranking need not be very precise, especially since tournament selection is used. Therefore, in case of such low fitness values, a single trial suffices ($R = 1$). For higher fitness values, the number of repetitions was set to $R = 10$. For a controller that has the potential to be the best of the population, the overall fitness was determined on the basis of $R = 100$ repetitions. Using this procedure the overall fitness of the fittest controller has a standard error of the mean of about 1.3, yielding an accuracy of about 2.5 fitness points (reliability of 95%; Cohen, 1995).

The validity of the evolved controllers was confirmed by testing them on a real Khepera robot. The controllers proved to be effective and efficient in letting a real Khepera robot push a circular box between walls. It is my opinion that this success is owing to the high amount of noise inherent in the simulation, which requires an evolved controller to be robust (Jakobi, 1997).

3.3.2 Results of the Box-Pushing Experiment

One experiment without doping and ten experiments with doping using various solutions were performed, and the overall fitness values were determined. For the doping

[2] I determined empirically that, in general, controllers with a fitness value of 250 or less worked well on the easy instances 0, 1, 4, 7 and 8, but were unable to deal with the hard instances 2, 3, 5 and 6.

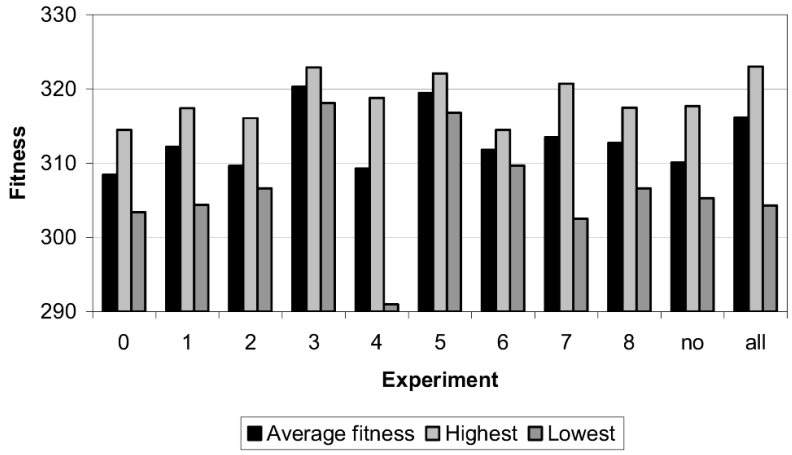

Figure 3.5: Fitness values of experiments with doping of a solution to a single instance ('0' to '8'), without doping ('no') and with doping of all solutions ('all'). From left to right, the bars represent the average, the highest, and the lowest fitness.

experiments DECA was applied by executing the six steps described in Subsection 3.1.3. Figure 3.4 shows examples of the trajectories of successful robots on the nine instances. To determine how doping with a solution to a hard instance compares to doping with a solution to an easy instance (instead of selecting a hard instance, as prescribed in step 1), I performed separate doping experiments with solutions to each of the nine instances (that vary from easy to hard). In addition, a doping experiment using solutions to all instances was performed. The average fitness values were obtained by averaging over the highest fitness values obtained in seven replications of each of the experiments.

I expected that doping with controllers trained on the hardest instances 3 and 5 to yield the best results. Indeed this was what I found. Figure 3.5 displays the results obtained with doping using controllers trained on a single instance (labelled '0' to '8'), without doping (labelled 'no') and with doping using controllers trained on all nine instances (labelled 'all'). Doping with controllers trained on instances 3 and 5 yield the best results (average fitness of 320.3 and 319.5, respectively), and the most consistent results (highest/lowest fitness values 322.9/318.1 and 322.1/316.8, respectively). Doping with controllers trained on all tasks yields better results than doping with controllers trained on instances that are easy or moderate (i.e., instances 0, 1, 2, 4, 6, 7, and 8). Presumably, the inclusion of solutions to the hardest instances contributes to the high fitness obtained in this case. It should be noted, however, that while doping with all instances gives the highest fitness, the results have a much higher variance than those obtained by doping with controllers trained on instances 3 and 5 (highest/lowest fitness values 323.0/304.3). I assume that the reason for this is that the evolutionary algorithm occasionally converges to a local optimum near to the optimal solution for instances other than 3 and 5.

Overall, these results show that on the box-pushing task DECA gives a significant improvement over non-doped evolutionary learning. The solutions found also perform considerably better than those found for the same problem by Sprinkhuizen-Kuyper (2001).

3.3.3 Discussion of the Box-Pushing Experiment

The box-pushing experiment was not specifically designed to test DECA. Yet, I was pleasantly surprised by the improved results obtained by applying DECA. Notwithstanding these results, it must be acknowledged that the box-pushing experiment task is of limited value for evaluating DECA. The reason is that it suffers from two main shortcomings, namely (i) the task is based on a stochastic simulation requiring many repetitions to obtain reliable results, and (ii) the lack of variety in possible instances precludes the assessment of the ability to generalise beyond the instances given (even though the controller's ability to generalise was demonstrated by applying it to a real Khepera).

I expected that the success of DECA can be generalised to other evolutionary control tasks. To support this expectation, I decided to evaluate DECA on a second control task, designed to deal with the limitations of the box-pushing experiment.

3.4 Food-Gathering Behaviour

The food-gathering experiment was designed to have the following two requirements: (i) the task should be deterministic, and (ii) the task should allow for generating instances with variable levels of difficulty. The food-gathering task is described in Subsection 3.4.1, the achieved results using DECA are presented in Subsection 3.4.2 and a discussion of the results is provided in Subsection 3.4.3.

3.4.1 The Food-Gathering Task

The food-gathering task is designed as follows. A rabbit is placed on a square two-dimensional grid of $N \times N$ cells. The rabbit can move by one step in each of the four orthogonal directions: north, east, south and west. The grid has periodic boundary conditions, i.e., it is defined as a torus. As illustrated in Figure 3.6, the rabbit's field of vision encompasses all cells that are within two moves from its current position. A cell may be empty, it may contain one or more carrots, or it may contain one or more poison bottles. If the rabbit enters a cell that contains c carrots, it removes (eats) all of them leaving an empty cell, and increases its score by c points. If the rabbit enters a cell with p poison bottles, it decreases its score by p points. In contrast to carrots, poison bottles are not removed from the grid when visited by the rabbit. In each experimental trial, a rabbit has to score as many points as possible within 100 moves. Initially, the rabbit is always positioned in an empty cell.

The rabbit is controlled by a neural network with thirteen inputs. Each input I is defined as the value of a cell visible to the rabbit (a shaded square in Figure

3.6; this includes the cell the rabbit currently occupies, which may contain poison). The magnitude $|I|$ of the input value represents the number of elements within the patch occupying the cell. The sign of the input indicates whether the patch contains carrots ($I > 0$) or poison bottles ($I < 0$). An empty cell is represented by zero input ($I = 0$). The network has four outputs, representing the four directions of movement of the rabbit. The rabbit moves in the direction corresponding to the output with the highest value.

For the training set grids were randomly generated with $N = 15$, a total number of carrots $C = 100$ and a total number of poison bottles P varying between 0 and 150. Carrots and poison bottles are clustered in small patches of one to five carrots or poison bottles per patch. The number of poison patches directly bordering a carrot patch also varies according to a density value d ($d \in \{0, 1, 2, 3, 4\}$). Arguably, the complexity of an instance is proportional to d and P, because an increased total number of poison bottles and an increased density of poison bottles adjacent to carrots make it harder for the rabbit to collect carrots without losing points.

Table 3.1 displays the twenty instances (numbered 0 to 19) in the training set in relation to the parameters d and P, including a qualification of their difficulty. For instances 5 to 17, the parameter d is defined as a range. Figure 3.7 shows six of the twenty instances that serve as the training set.

To assess the generalisation performance of evolutionary designed rabbits, an extensive test set of a hundred instances was generated, comprising five subsets of twenty randomly-generated instances each. The instances within each subset were generated according to the same values of d and P as specified in Table 3.1.

The fitness F of a controller (or rabbit) is defined as the average score on the twenty instances of the training set. Since each instance contains 100 carrots, an up-

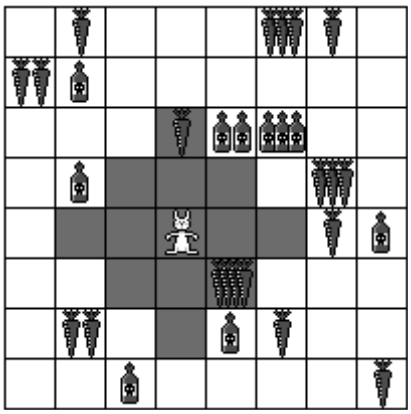

Figure 3.6: Part of the grid defined as the environment of the rabbit. The environment contains food (carrots) and danger (poison bottles). The rabbit's field of vision consists of all cells (squares) that can be reached in a maximum of two moves (the shaded squares in the image), i.e., the Manhattan distance = 2.

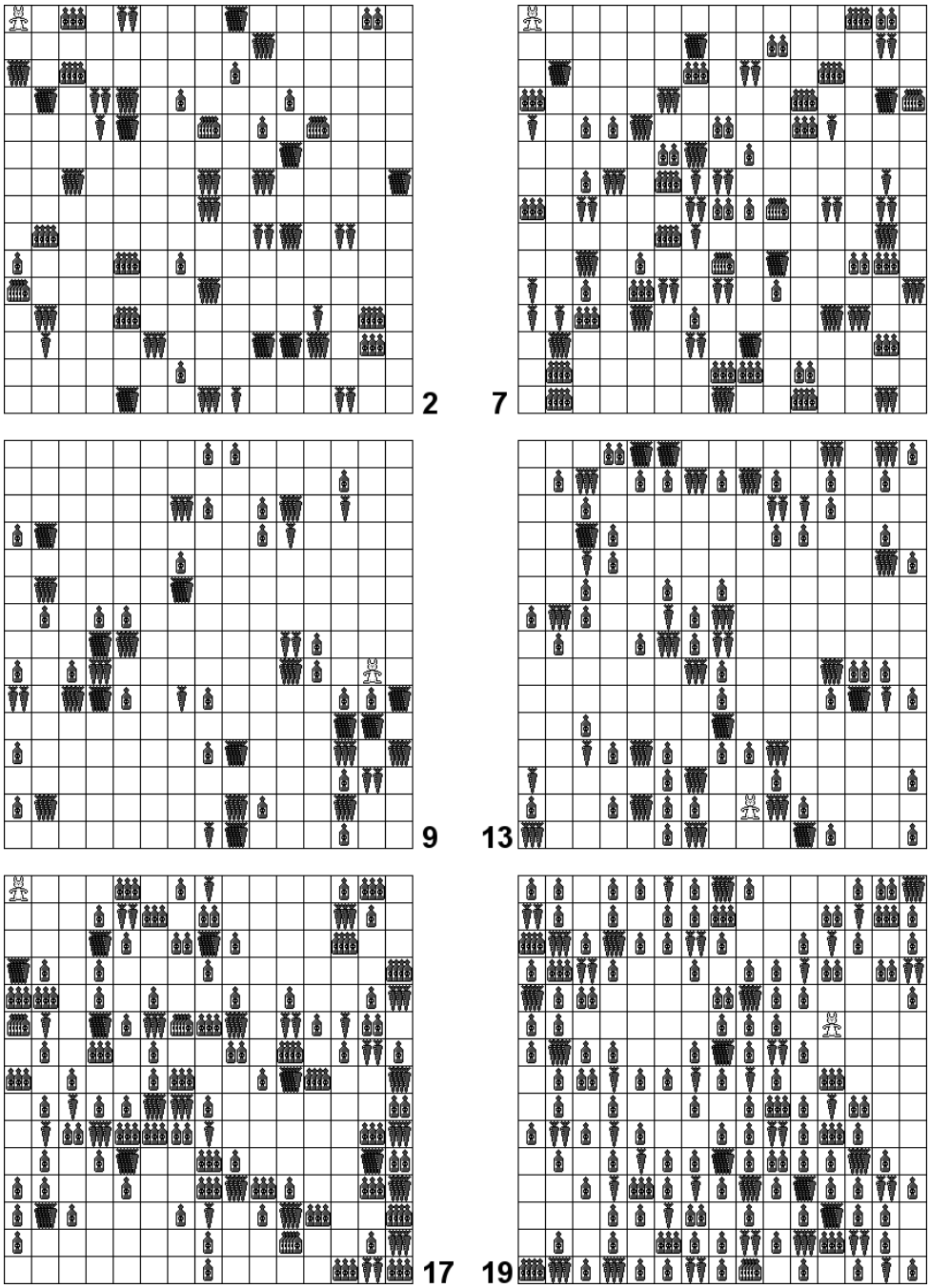

Figure 3.7: Six of the twenty instances in the training set of the food-gathering experiment.

instance	d	P	difficulty	instance	d	P	difficulty
0	0	0	very easy	10	1–2	50	medium
1	0	25	very easy	11	1–2	100	medium
2	0	50	easy	12	1–2	150	hard
3	0	100	easy	13	2–3	50	medium
4	0	150	medium	14	2–3	100	hard
5	0–1	25	easy	15	2–3	150	hard
6	0–1	50	easy	16	3–4	100	hard
7	0–1	100	medium	17	3–4	150	very hard
8	0–1	150	medium	18	4	100	very hard
9	1–2	25	easy	19	4	150	very hard

Table 3.1: Specification of the twenty instances (numbered 0 to 19) used in the food-gathering experiments in relation to the density value d and the number of poison bottles P. In all instances $C = 100$.

per bound to the fitness is 100. In most instances it is impossible to reach this upper bound, because even without poison patches, usually the shortest path connecting all carrot patches in the grid is longer than 100 steps.

3.4.2 Results of the Food-Gathering Experiment

For the food-gathering experiment two series of tests were compared. In the first series, the evolutionary algorithm discussed in Section 3.2 was used to evolve, in 30 generations, a neural controller for the rabbit, with a fitness function defined as the average score of the controller on the twenty grids in the training set. In the second series DECA was applied, as follows. First, a good controller for a single instance was evolved. Then a neural controller was evolved with the overall fitness function in 27 generations, using an initial population doped with the solution found for the single instance. The reason for using 27 (rather than 30) generations for the evolution with the overall fitness function was to ensure that the computational resources used for both series of experiments were approximately equal.

I decided to use instance 17 as the hard instance to develop a good controller for doping. In this instance $P = 150$ and $d = 3$–4. I preferred instance 17 over the seemingly harder instance 19 (with $P = 150$ and $d = 4$), because in instance 19 all carrot patches are completely surrounded by poison. I suspected this would reduce the complexity of the task, because it would be impossible for a controller to avoid damage to get to carrots. Therefore, damage avoidance is of less importance for instance 19 than for instance 17.

$R = 100$ repetitions of each of the series of tests were run. Of each of the tests, the controller with the highest fitness on the training set containing twenty grids, was used as the solution found. Then this controller was evaluated on the test set containing 100 grids. In the statistical analysis the fitness of a controller was defined

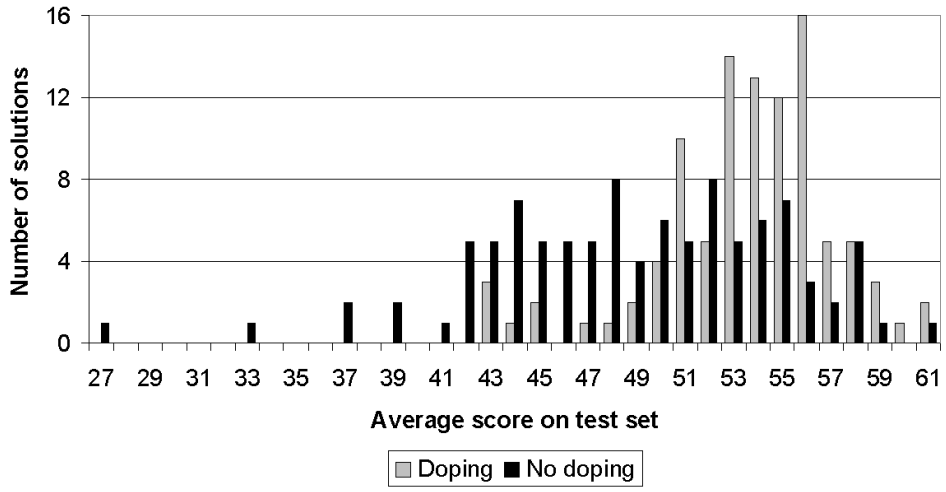

Figure 3.8: Comparison of the scores of 100 tests with doping and 100 tests without doping in the food-gathering experiment.

as its score on the test set. In Figure 3.8 the histograms of the experiments with and without doping are displayed.

As is evident from the histograms, the experiments with doping tend to give better solutions than those without doping. The minimum score achieved without doping is 27, while the minimum score achieved with doping is 43. The highest score achieved is 61 both with doping (twice) and without doping (once). For doping, the bulk of the scores range from 50 to 60, whereas the bulk of the scores obtained without doping are more widely distributed, namely between 40 and 60.

Without doping, the score of evolutionary-designed controllers averaged over 100 experiments is 48.9 with a standard error of the mean of 0.6. With doping, the score averaged over 100 experiments equals 53.6 with a standard error of the mean of 0.4. From these numbers it can be concluded that the results achieved with doping are significantly better than those achieved without doping (reliability $> 99.9\%$; Cohen, 1995).

3.4.3 Discussion of the Food-Gathering Experiment

The food-gathering task is deterministic and allows for the generation of novel instances. Both characteristics offer the advantage that the effect of doping can easily be assessed. Clearly, the results show that doping is useful for enhancing the quality and generalisation performance of evolutionary-designed controllers.

To illustrate the type of solutions obtained, a striking example is presented in Figure 3.9. It shows a path followed by a successful rabbit (controller) on a hard instance ($P = 150$ and $d = 3$–4). Despite the ability to move in four directions,

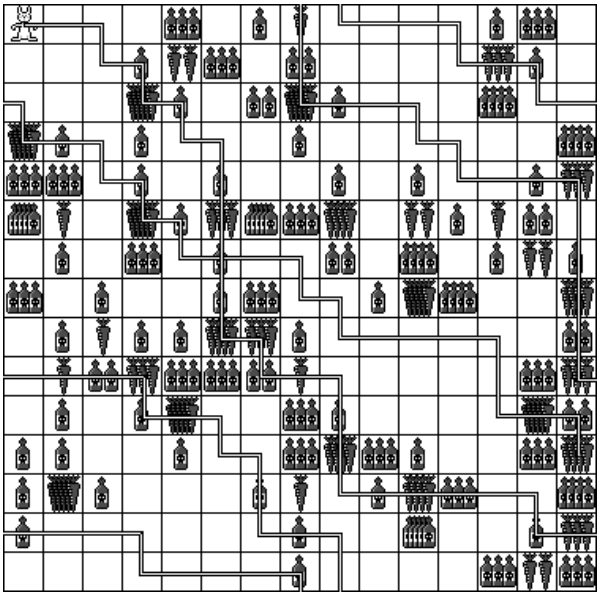

Figure 3.9: An example of the path taken by a successful rabbit in a hard environment.

the rabbit moves to the east and south only. In a post-hoc analysis of successful controllers, I noticed that two of their four outputs (namely one of the two longitude outputs and one of the two latitude outputs) were disconnected. Constraining the movement to two orthogonal directions prevents rabbits from moving in circles, which leads to suboptimal performance.

3.5 Discussion

The application of DECA to two different tasks showed the feasibility of the DECA approach. In this section DECA will be discussed in more detail. Subsection 3.5.1 provides insight into why the doping effect occurs. Doping is compared to hill-climbing in Subsection 3.5.2. I discuss five search techniques that provide an alternative approach to deal with the problem of hard instances, namely (i) multitask learning (3.5.3), (ii) multi-objective learning (3.5.4), (iii) boosting (3.5.5), (iv) island-based evolutionary learning (3.5.6), and (v) constraint-satisfaction reasoning (3.5.7). Finally, Subsection 3.5.8 discusses how DECA can be applied to the evolutionary learning of game AI.

Note that I do not claim that evolutionary learning of a neural controller with DECA provides the *best* solutions for the problem domains discussed in this chapter. Other techniques that use a training set, such as reinforcement learning, may

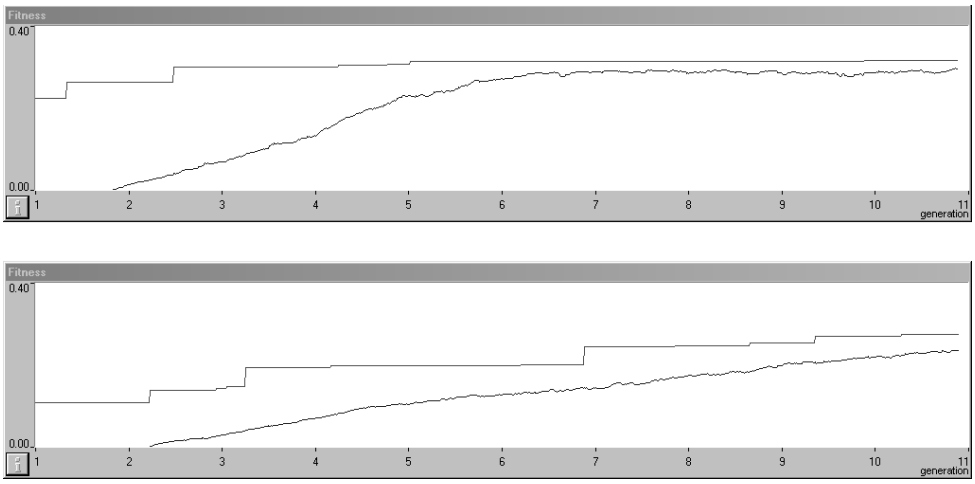

Figure 3.10: Typical developments of fitness for evolutionary learning with doping (top graph) and without doping (bottom graph). In both graphs the fitness (divided by 1000) is plotted against the generation. The top curve in each of the graphs shows the maximum, and the bottom curve the average fitness in the population.

generate solutions of a quality comparable to, or even higher than the quality of the solutions discovered by evolutionary learning. The point is that these other techniques are also likely to discover better solutions when the doping effect is taken into account. Therefore, I refrain from discussing such alternative techniques.

3.5.1 Explanation of the Doping Effect

Why is DECA a successful strategy? Below I attempt to provide a qualitative explanation for the success of doping.

The search space of task control problems is spanned by the adaptable parameters defining the controllers, i.e., by the connection weights in the neural networks. Hence, the dimensionality of the search space is defined by the number of adaptable parameters specifying the controllers (the dimensionalities of the box-pushing and food-gathering controllers are 81 and 92 respectively). As stated in Subsection 3.1.3 I assume that the high-dimensional search space contains abundant regions where solutions to easy instances are found, but only a few small regions where solutions to hard instances reside. Because the hard instances encompass many, if not all of the difficulties posed by the environment, a solution that applies to instances of arbitrary complexity is likely to be found relatively near to a hard-instance region. Hence, doping the initial population with a solution specialised to hard instances leads to good generalised solutions.

The explanation is supported by the development of the fitness of evolution processes with and without doping. In Figure 3.10 the developments of fitness in

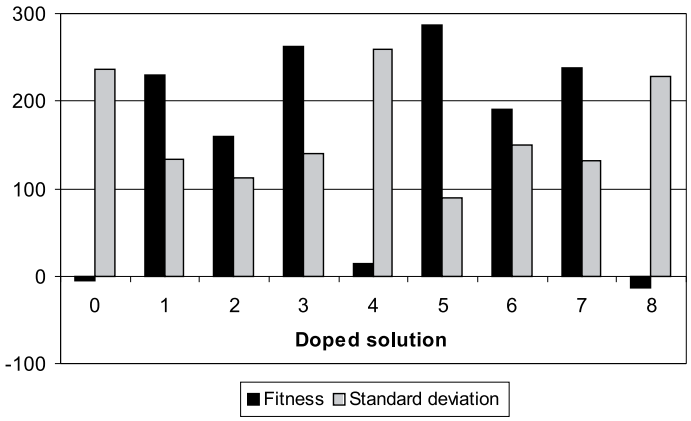

Figure 3.11: The fitness of doped solutions to a single task, tested on all instances, averaged over 100 tests. For doping with solutions to each of the nine instances (0 to 8), the graph shows the fitness (the left, black bar), and the standard deviation (the right, shaded bar).

the evolution of a box-pushing task with doping (the upper graph) and without doping (the lower graph) are compared. While these are only two examples, I found that they are typical for all tests. With doping the fitness of the best controller in the population starts between 200 and 250. Within one or two generations, the fitness jumps to around 300. After that, the fitness slowly increases towards a value around 320. Without doping, the fitness starts anywhere between 0 and 200. Initially, the fitness increases quickly to a value between 200 and 250. After that, the fitness progresses slowly towards a value of about 310. These different patterns of development, in particular the quick rise in fitness at the start of the doped evolution process, suggest that DECA takes the best available solution (the doped one) and adapts it to handle the other instances.

Further support for the explanation is found in experiments that indicated that solutions to hard instances also perform reasonably well on the easy instances, whereas the same is not true the other way round. For the box-pushing task this is illustrated in Figure 3.11. It shows, for each of the doped controllers used in the box-pushing experiments, the fitness and standard deviation on all instances, averaged over 100 tests. Controllers evolved on the hardest instances 3 and 5 yield the highest fitness on all other instances, combined with the lowest standard deviation.

To provide solid evidence for the explanation, first the key assumption in the explanation for DECA's success, namely the supposed asymmetry of the search space with respect to easy and hard solutions, needs to be verified. Moreover, the belief that solutions to hard instances encompass characteristics of solutions to easy instances is a major ingredient for DECA's success, must be confirmed. A possible approach to this future research is testing DECA on a variety of benchmark

problems, designed to exhibit specific characteristics with respect to the architecture of the search space, and with respect to overlapping features between instances. Tracing the lineage of the best evolved solutions to the benchmark problems, to determine whether and how they include doped solutions in their ancestry, will be a key activity in understanding the factors responsible for DECA's success.

3.5.2 DECA and Hillclimbing

Since the explanation for the doping effect states that the evolution process adapts the doped solution to become a general solution, the question may be posed whether DECA may be combined with hillclimbing. Given a doped solution, hillclimbing may represent a good alternative to standard evolutionary learning to obtain good results. I believe, however, that hillclimbing is not a good alternative to evolutionary learning in DECA for the following reason. While the generalised solution may be in the vicinity of the solution to a hard instance, it is unlikely that it is in the vicinity of all dimensions of the hard instance. Sometimes, adapting the solution to the hard instance to generalise over all instances requires large steps in one or a few dimensions of the search space. In contrast to hillclimbing, evolutionary algorithms are capable of doing that.

Of course, the nature of the search space depends on the type of problem. Hence, hillclimbing may yield good results in some cases, which should be examined in future work. Montana and Davis (1989) support my line of reasoning in this respect, by stating that hillclimbing does not work well for neural network training, since it tends to force convergence to a local optimum instead of a global optimum. They recommend using hillclimbing only in those cases where the best solution achieved is close to the global optimum.

3.5.3 DECA and Multitask Learning

The principal goal of multitask learning is to improve generalisation performance of a controller on a task, by leveraging information obtained from controlling related tasks. It does this by training tasks in parallel using a shared representation. Caruana (1997) claims, and shows empirically, that it is more difficult to train a controller on an isolated, difficult task, than it is to train a controller on a combination of related tasks that includes the difficult one. At first glance, this seems to be in conflict with my claim, that doping with a controller for a hard instance generalises better than doping with a controller for an easy instance.

As Caruana (1997) explains, the 'related tasks' used in multitask learning are not so much various instances, but simpler subtasks. With DECA the task is the same for each instance, only the environment differs. The claims Caruana (1997) makes about multitask learning are, therefore, not in conflict with the claims I make about DECA. Moreover, I suspect that multitask learning actually suffers from the hard-instances problem, because it deliberately focusses on easier tasks before tackling a hard one. It does that for a good reason, namely that the hard task cannot be solved directly. Obviously, DECA is not intended to deal with these 'unsolvable' tasks.

Louis and Li (1997) use an approach to multitask learning reminiscent of DECA. They evolve solutions to subtasks and use those to dope the initial population of an evolution run that solves the overall task. They discovered that doping with the best solution to each of the subtasks actually results in worse overall solutions than starting with a randomly initialised population. However, doping with solutions to subtasks that also give good results on the overall task, leads to significantly better solutions than achieved with a randomly initialised population. This result supports my suggestion in Subsection 3.5.1, where it is stated that the doping effect results from solutions to hard instances encompassing characteristics that are needed to solve the easier instances.

It is possible that a combination of multitask learning and DECA, where controllers for hard instances of the subtasks are doped, may improve the performance of either technique alone. This is an interesting notion that warrants exploration in future work.

3.5.4 DECA and Multi-Objective Learning

Multi-objective learning aims to find a solution that performs well with regard to all individual objectives in a set of (often) conflicting objectives (Van Veldhuizen and Lamont, 2000). The main problem of multi-objective learning is that it tends to get stuck in a local minimum once a solution is found for one of the objectives. It is generally appreciated (Horn, 1997; Van Veldhuizen and Lamont, 2000) that a successful Multi-Objective Evolutionary Algorithm (MOEA) needs a secondary population to store Pareto-optimal solutions (e.g., solutions to single objectives), sometimes actually involving the secondary population in the evolution process.

The instances used in the DECA experiments bear some resemblance to the objectives in multi-objective learning. Interpreting the task instances as different objectives, multi-objective learning techniques can be applied to the problem of hard instances, since they seek a balance between several conflicting objectives (Van Veldhuizen and Lamont, 2000). However, the instances in the DECA experiments do not represent different objectives, but different incarnations of the environment, while the task to be performed is the single objective. Furthermore, the environments are mostly not in conflict with each other. Since, in general, multi-objective learning techniques are geared towards conflicting objectives, they do not exploit the similarity between the various environments. Therefore, I believe DECA to be better suited for handling the particular domain of task control problems. This belief must be tested in future work.

3.5.5 DECA and Boosting

Boosting (Schapire, 2002) is a learning method, usually employed to design classifiers, that assigns each sample in the training set a weight. At the beginning all weights are equal, but over time the samples that are handled badly receive higher weights than those that are handled well, so that the focus of the learning shifts to the harder samples. If the explanation we gave in Subsection 3.5.1 for the doping

effect is correct, boosting will at least give evolutionary algorithms a better chance to escape from local optima where easy instances are handled well but hard instances are not. However, it does not have DECA's advantage of starting in a local optimum for a hard instance, in the neighbourhood of which a global optimum should be located. I therefore expect that controllers created with boosting on average will be inferior to those created with DECA. Clearly, this expectation requires empirical validation, which is considered future work.

3.5.6 DECA and Island-Based Evolutionary Learning

Evolutionary algorithms are inherently parallel. On multi-processor computers this is commonly exploited by dividing the population into smaller sub-populations, each of which is handled by a different processor. The sub-populations are often referred to as 'islands' (Goldberg, 1989). On each island the population is evolutionary trained on a particular task. The islands exchange genetic material on a regular basis. Apart from enabling parallel processing, the islands may converge to different solutions. The exchange of genetic material might result in an overall solution that combines the best of the island-based solutions.

Island-based evolutionary learning (Spronck, Sprinkhuizen-Kuyper, and Postma, 2001b) is an attempt to exploit the principles behind parallel evolutionary algorithms to solve the problem of hard instances. The basic idea of island-based evolutionary learning is to distribute the population evenly over a few islands, whereby each island is assigned a different task instance. After all island populations have converged to a solution to their assigned task, a new population of the best solutions of each of the islands and a number of random solutions is created. A conventional evolutionary algorithm is applied to this new population that is trained to deal with all instances. The idea is that the evolution combines genetic material developed using single instances to solve the general task.

Clearly, island-based evolutionary learning may very well be applied to the problem of hard instances. However, empirical studies, using the box-pushing task, have revealed that island-based evolutionary learning tends to generate solutions that perform well on the hard instances (even better than when a regular evolutionary algorithm is applied), but show an inferior performance on the easy instances. As a consequence, a gain in overall fitness is not obtained (Spronck *et al.*, 2001b). Furthermore, since island-based evolutionary learning evolves a separate solution for all instances, the computational time required by the island-based evolution process is much larger than the computational time required by DECA.

3.5.7 DECA and Constraint-Satisfaction Reasoning

Constraint satisfaction reasoning (CSR) deals with problems where the solution has to satisfy a given set of restrictions or constraints (Tsang, 1993). A solution is invalid unless it fulfils all the constraints. Hence, in CSR the problem is to find a solution that takes into account all constraints rather than one that addresses some of the constraints. Interpreting the instances as constraints, CSR seems applicable

to alleviate the problem of hard instances. However, CSR cannot be readily applied to the problem. The reason is that in CSR all constraints must be strictly satisfied, whereas in task learning it suffices if the instances are handled reasonably well.

3.5.8 DECA and Game AI

Both the box-pushing task and the food-gathering task have strong ties to tasks that agents have to solve in modern computer games. The box-pushing task concerns robot control in a noisy environment, which can be compared to, for instance, controlling a race car in a racing game (Pyeatt and Howe, 1998), or controlling a soccer-playing agent in a sports game (Van Rijswijck, 2003). The food-gathering task concerns effective path-finding in an environment filled with dangers and rewards, which can be compared to, for instance, army movement in a strategy game (Buro, 2003b), or maze-traversing in an arcade game (Ledwich, 2003). In games, the game AI is responsible for controlling the agents. The results achieved with DECA indicate, that when game AI is created by an evolutionary algorithm, doping the initial population with game AI that has been evolved on the hardest agent task, is likely to result in game AI that is more effective than when evolved using a randomly-initialised population. This conjecture will be used in Chapter 6.

Despite the similarities between the two experimental environments used in this chapter, and some types of agents in games, the question remains whether the learning techniques used, evolutionary algorithms and neural networks, are suitable for game AI. Spronck *et al.* (2002) provided an answer to that question, stating that they are suitable for offline learning of game AI, but not for online learning of game AI. Chan *et al.* (2004) and Madeira, Corruble, Ramalho, and Ratitch (2004) reached similar conclusions with respect to evolutionary algorithms. Chapter 4 will further explore this subject.

3.6 Chapter Summary

In this chapter the problem of hard instances was identified, and the DECA approach was proposed to deal with it. In particular, it was demonstrated how doping an initial population with a solution to a single hard instance improved the performance on two quite different tasks. Given the results on the box-pushing and food-gathering tasks it may be concluded that the problem of hard instances is alleviated by the application of DECA. Moreover, compared to 'regular' evolutionary algorithms, solutions discovered by DECA not only perform better on hard instances, but also perform better overall, i.e., achieve a significantly higher average fitness. With respect to games, this means that, when evolutionary algorithms are used to create the game AI, doping the initial population can be expected to generate better results than when using a randomly-initialised population.

Chapter 4

Evolutionary Game AI

The art of progress is to preserve order amid change
and to preserve change amid order.
— Alfred North Whitehead (1861–1947).

In Chapter 3 it was shown that evolutionary algorithms can improve the behaviour of agents for task control problems. The present chapter[1] discusses evolutionary game AI, i.e., game AI that employs evolutionary algorithms. The purpose of using evolutionary algorithms in game AI is providing a high-entertainment value for human players by evolving challenging agent tactics. Section 4.1 empirically investigates offline evolutionary game AI, that has the ability to pinpoint potential weaknesses in the agent's behaviour, and to design new tactics. Section 4.2 empirically investigates online evolutionary game AI, that has the ability to improve game-playing tactics against a specific human player. Section 4.3 provides a general discussion of evolutionary game AI. A summary of the chapter is provided in Section 4.4.

4.1 Offline Evolutionary Game AI

Offline evolutionary game AI controls agents that are in competition with agents that employ existing (usually manually-designed) game AI. Offline evolutionary game AI has two applications: (i) to detect exploits in the existing game AI (Spronck *et al.*, 2002; Chan *et al.*, 2004), and (ii) to discover new tactics that can be used against the existing game AI (Spronck *et al.*, 2002; Madeira *et al.*, 2004). Note that, because human players are only indirectly involved when offline learning takes place, it is infeasible to use offline evolutionary game AI to adapt the agent's behaviour to specific human-player tactics (Madeira *et al.*, 2004). To investigate the effectiveness of offline evolutionary game AI, I tested it on a duelling task in a small strategy game called PICOVERSE. The approach used consisted of the following four steps.

[1] This chapter is based on two papers. Section 4.1 on offline evolutionary game AI is based on a paper by Spronck, Sprinkhuizen-Kuyper, and Postma (2003a). Section 4.2 on online evolutionary game AI is based on a paper by Bakkes, Spronck, and Postma (2004).

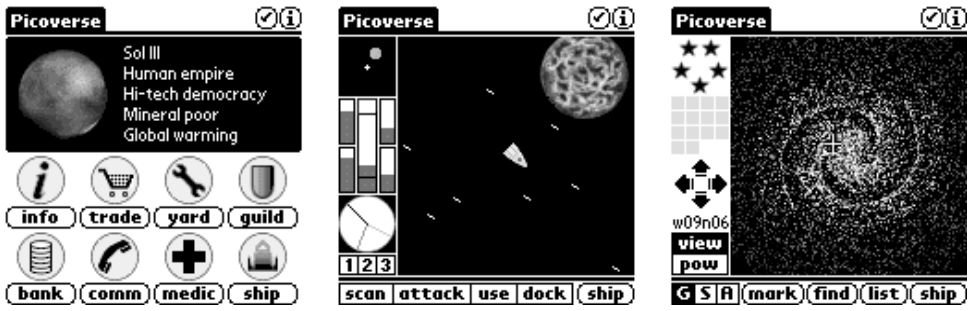

Figure 4.1: PICOVERSE.

1. *Evolution:* Evolving duelling behaviour that is successful against the manually-designed game AI.

2. *Analysis:* Observing and analysing the evolved duelling behaviour, to gain insight into which areas of the manually-designed game AI can be improved.

3. *Derivation:* Deriving potential improvements for the manually-designed game AI.

4. *Validation:* Implementing the potential improvements in the manually-designed game AI, and repeating the *Evolution* step to investigate their effect.

This section describes the duelling task (4.1.1), the experimental procedure (4.1.2), the results of the *Evolution* step (4.1.3), the results of the *Analysis* step (4.1.4), the results of the *Derivation* step (4.1.5), the results of the *Validation* step (4.1.6), and a discussion of the results (4.1.7).

4.1.1 The Duelling Task

PICOVERSE, illustrated in Figure 4.1, is a strategy game for the Palm (handheld) computer. PICOVERSE's design was inspired by the classic game ELITE by D. Braben and I. Bell (Spufford, 2003). It was developed for two reasons: (i) to support and illustrate views on the design of complex Palm games (Spronck and Van den Herik, 2003), and (ii) in the present context, to investigate the application of machine-learning techniques to improve game AI.[2]

In PICOVERSE, a human player controls a spaceship (henceforth called the 'player's ship'). In the game, the player's ship may encounter computer-controlled enemy ships, and combat may ensue between the player's ship and the enemy ships. All ships are equipped with laser guns, and are protected from destruction by their hulls. Hull strength decreases when a hull is hit by laser beams fired from the laser

[2]Because of time constraints, in 2003 developments on PICOVERSE were put on hold, to be continued at a later date.

guns. The strength of laser guns and the hull strengths vary from ship to ship. A ship is destroyed when its hull strength is reduced to zero. Ships are controlled by changing their acceleration (which increases or decreases velocity), and by changing their rotation (which steers a ship in a different direction). While the relative strength of laser guns and relative hull strength of battling ships are important factors in deciding the outcome of combat, ships have a chance to flee from a battle even when they are overpowered, provided they are equipped with fast and flexible drives. However, attempting to flee is a risky action, because a fleeing ship is unable to counterattack. The reason is that, to flee, a ship must turn its back to its attacker, and laser guns can only fire within a 180-degree arc at the front of a ship.

As is usual for modern games, the computer-controlled enemy ships are programmed manually. Upon detecting the player's ship, an enemy ship will turn towards it and attempt to catch up with it. When the player's ship is within laser range of an enemy ship, the enemy ship will fire its lasers. It will also attempt to keep the player's ship within laser range, by matching the speed of the player's ship. To evoke a suspension of disbelief, an enemy ship will attempt to escape from a duel that it is bound to lose, rather than continue fighting until it is destroyed. This fleeing behaviour is implemented as follows: if the ratio of the current and maximum hull strength of the enemy ship is lower than the corresponding ratio of the player's ship, the enemy ship attempts to flee by turning around and flying away at maximum speed. This simple yet effective behaviour mimics a basic tactic often used in games. It makes the opponent intelligence for PICOVERSE non-trivial, despite the relatively low level of complexity compared to state-of-the-art games.

Figure 4.2 illustrates the manually-programmed behaviour. The duelling spaceships are represented by the small circles. A ship's direction is indicated by a line inside the circle, and its speed is indicated by the length of the line extending from the ship's 'nose'. The dotted arc indicates the laser range. The player's ship is fixed at the centre of the screen and directed to the right. During the sequence shown in Figure 4.2 it remains stationary. From left to right, top to bottom, the pictures demonstrate the following six events: (i) The two ships starts within viewing range of each other (the viewing range of the player's ship is delimited by the large circle). (ii) The computer-controlled enemy ship moves towards the player's ship. (iii) The ships bump head-on into each other, which reduces the speed of both ships to zero. Both ships are firing their lasers. (iv) The enemy ship has determined it should flee and turns around. (v) The enemy ship flees. (vi) The enemy ship escapes by leaving the viewing range of the player's ship.

The duelling task entails designing successful behaviour for the player's ship against the enemy ships. Successful behaviour for the player's ship can be used to detect weaknesses in the manually-programmed behaviour of the enemy ships, and to design completely new tactics.

4.1.2 Experimental Procedure

Offline evolutionary game AI was used to solve the duelling task experimentally. The success of the experiments with agents in game-like environments (Chapter 3) war-

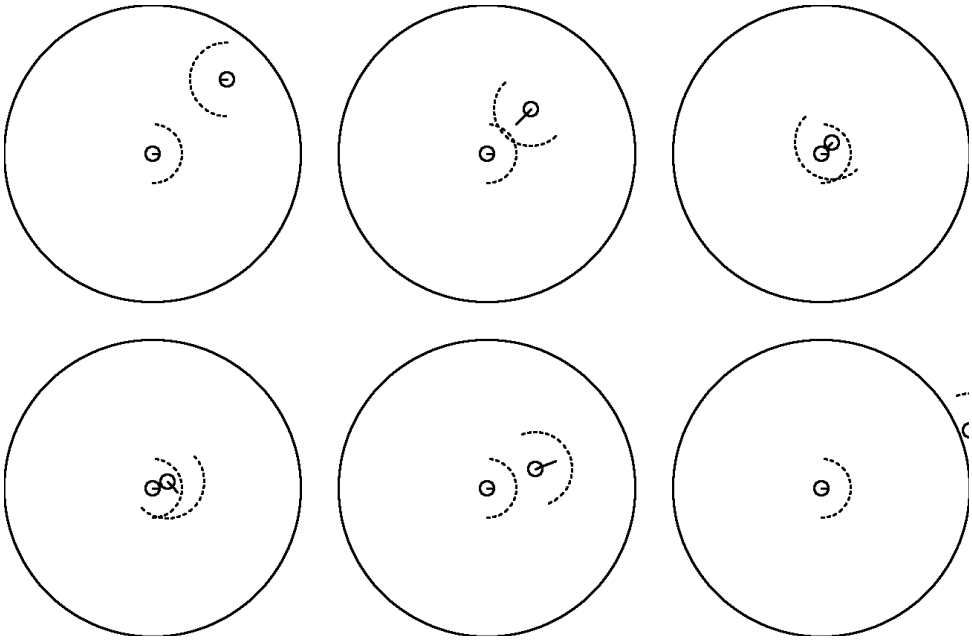

Figure 4.2: Manually-programmed behaviour for the PICOVERSE computer-controlled ships.

ranted a similar approach to the duelling task. The duelling task was implemented in the ELEGANCE environment (2.1.4). ELEGANCE uses evolutionary learning to evolve solutions for 'plants'. Below, I describe four elements of the experimental procedure: (i) the plant implementation, (ii) the neural-network controller, (iii) the evolutionary algorithm, and (iv) the fitness function.

The first element of the experimental procedure is the duelling-task plant. The duelling-task plant represents a player's ship, in a series of combat situations with an enemy ship. The player's ship uses dynamically determined behaviour, and is called the 'dynamic ship'. The enemy ship uses manually-programmed, static game AI (as described in Subsection 4.1.1), and is called the 'static ship'. For both ships, laser guns fire automatically at appropriate times, and need not be controlled. Thus, plant control consists of setting the acceleration and rotation values for the dynamic ship.

The movement of the ships is turn-based. Movements are executed in an alternating sequence. The dynamic ship is allowed to move first and the static ship is always allowed a last move even if its hull strength is reduced to zero. For two reasons a turn-based approach was preferred over a simultaneous approach to the combat sequences: (i) a turn-based approach is used in a number of popular strategy games, and (ii) a turn-based approach is computationally significantly cheaper than

a simultaneous approach, which is an important consideration for time-intensive evolutionary-learning experiments.

The second element of the experimental procedure is the neural-network controller. In the experiments, the dynamic ship is controlled by a neural network, i.e., the game AI of the dynamic ship is implemented by a neural-network controller. To reduce the number of required neural-network inputs, coordinates are used relative to the dynamic ship, i.e., the 'game world' is moved so that the dynamic ship is located at its centre, and rotated so that the dynamic ship's 'nose' is pointed at an angle of zero degrees.

Ten neural-network inputs were used to represent the environment. Four inputs represent characteristics of the dynamic ship: (i) the laser-gun strength, (ii) the laser-gun range, (iii) the hull strength, and (iv) the speed. Five inputs represent characteristics of the static ship: (i) the location direction of the static ship relative to the dynamic ship, (ii) the distance between the static ship and the dynamic ship, (iii) the current hull strength, (iv) the flying direction, and (v) the speed. The tenth input is a random value, to allow the dynamic ship an element of randomness in its decisions. The neural network has two outputs, namely the acceleration and rotation of the dynamic ship. The hidden nodes in the network have a sigmoid activation function. The outputs of the network are scaled to ship-specific maximums.

The third element of the experimental procedure is the evolutionary algorithm. The parameters for the evolutionary algorithm were determined during a few trail runs. For the evolutionary algorithm, the population size was equal to 200 and real-valued weights were used. Experiments were allowed to continue for 50 generations. The following six genetic operators were employed.

- *Uniform crossover*: Child chromosomes are created by copying each allele from one of two parents, each parent having a 50 per cent chance of being selected for each allele (Goldberg, 1989).
- *Biased weight mutation* (Montana and Davis, 1989): Child chromosomes are copies of parent chromosomes, with each weight having a 10 per cent chance to be mutated by adding a random value selected from the range $[-2.0, 2.0]$.
- *Nodes crossover* (Montana and Davis, 1989): Child chromosomes are created by copying each of their nodes (including their input connections) from one of two parents, each parent having a 50 per cent chance of being selected for each node.
- *Node existence mutation* (Spronck, 1996): Child chromosomes are copies of parent chromosomes, with a 75 per cent chance of having all incoming and outgoing connections of one randomly-selected node being removed, and a 25 per cent chance of having all absent connections of a randomly-selected node being activated.
- *Connectivity mutation* (Spronck, 1996): Child chromosomes are copies of parent chromosomes, whereby each connection has a probability of 10 per cent to switch from being connected to being disconnected and vice versa.

- *Randomisation*: A random new child chromosome is generated to prevent premature convergence.

During evolution, one of these six operators was selected at random. For the crossover operators, I decided to add both children to the population. To alleviate the problem of competing conventions (2.1.3) the hidden nodes of the parents were rearranged to make their signs match (insofar as possible) before a crossover took place (Thierens *et al.*, 1993). Newly generated individuals replaced existing individuals in the population, while taking into account elitism. Size-3 crowding (Goldberg, 1989) was used as replacement policy. For the selection process, size-2 tournament selection was used (Goldberg and Deb, 1991).

The fourth element of the experimental procedure is the fitness function. The fitness of the dynamic-ship controller, with a value in the range $[0, 1]$, is defined as the average result on a training set of fifty duels between the dynamic ship and the static ship. The starting distance between the two ships in all of the 50 training-set cases is in the range $[80, 125]$. Each duel lasts $T = 50$ time steps. To ensure equal opportunities for the dynamic ship and the static ship to achieve high fitness, each duel in which the ships start with different characteristics is followed by a duel in which the characteristics are exchanged between both ships. At time step t the fitness is defined as in the following equation.

$$F_t = \begin{cases} 0 & D_t \leq 0 \\ \dfrac{S_0 D_t}{S_0 D_t + D_0 S_t} & D_t > 0 \end{cases} \tag{4.1}$$

In this equation, D_t and S_t are the hull strengths of respectively the dynamic ship and the static ship at time t. The fitness is 0.5 if both ships remain passive or are damaged for an equal percentage. If the static ship is damaged for a larger percentage than the dynamic ship, the fitness is greater than 0.5, and if the reverse is true (or when the dynamic ship is destroyed) the fitness is smaller than 0.5. Consequently, the fitness function favours attacking if it leads to victory, and favours fleeing otherwise. The overall fitness F for a duel is determined as the average of the fitness values at each time step, i.e., $F = \sum_{t=1}^{T} \frac{F_t}{T}$.

4.1.3 Evolving Successful Duelling Behaviour

An experiment with offline evolutionary game AI was performed, with the purpose of evolving duelling behaviour that is successful against the manually-designed game AI, described in Subsection 4.1.1. Since the experiment was executed using ELEGANCE, a neural network was used to implement the evolved behaviour. Different neural-network architectures may yield different results. For lack of insight into which neural-network architecture gives the best results for the duelling task, I decided to test seven different architectures, which are listed in Table 4.1.

The question should be answered how successful duelling behaviour can be recognised. It can be argued that a neural-network controller with a fitness value > 0.5

Neural network type	Hidden layers	Hidden nodes	Tests	Average fitness	Lowest fitness	Highest fitness
Recurrent	1	5	5	0.516	0.459	0.532
Recurrent	1	10	5	0.523	0.497	0.541
Recurrent	2	10	7	0.504	0.482	0.531
General feed-forward	n/a	7	5	0.472	0.382	0.527
Layered feed-forward	2	10	5	**0.541**	0.523	**0.579**
Layered feed-forward	2	20	8	**0.537**	0.498	**0.576**
Layered feed-forward	3	15	7	0.515	0.446	**0.574**

Table 4.1: Results achieved in the duelling-behaviour experiment, for seven different neural-network controller architectures.

performs better than the static ship's game AI. But how high can we expect the fitness actually to become? To provide an answer to that question, I calculated the fitness of a dynamic ship that is stationary, i.e., that will fire its laser guns at the static ship when appropriate, but that will not accelerate or rotate. I found that, on the training set, a stationary dynamic ship achieves a fitness of 0.362. If the fitness for the static ship is calculated according to formula 4.1, the static ship's fitness is $1 - F$, where F is the dynamic ship's fitness. Since it is reasonable to assume that the static ship performs better than a stationary ship, a fitness of $1 - 0.362 = 0.638$ can be considered an upper bound to the fitness of the dynamic ship's controller.

Table 4.1 presents the results achieved for evolving neural-network controllers for the dynamic ship. For each of the neural-network architectures tested, from left to right, the columns indicate (i) the neural-network architecture, (ii) the number of hidden layers, (iii) the number of hidden nodes (the hidden nodes are evenly distributed over the hidden layers), (iv) the number of tests, (v) the average fitness value, (vi) the lowest fitness value achieved, and (vii) the highest fitness value achieved. The best results for the average and highest fitness values achieved are printed in boldface. Two conclusions are derived from Table 4.1.

First, it is evident that, in this environment, two-layered feed-forward networks outperform all other networks in terms of both average and maximum fitness values. The network with five nodes in each hidden layer did not score significantly better than the network with ten nodes in each layer.

Second, a layered feed-forward neural network with 10 hidden nodes in two layers achieved a fitness of 0.579. Compared to the theoretical upper bound of 0.638, a fitness value of 0.579 indicates very successful duelling behaviour.

It should be noted, that from the perspective of game-play experience, the fitness rating as calculated in the experiment is not as important as the objective result of a fight. A fight can end in a victory, a defeat, or a tie.[3] For the best controller

[3] A tie means that both ships survive the encounter. It does not mean that both ships are destroyed. The destruction of both ships is considered to be a loss for the dynamic ship.

evolved, we found that 42 per cent of the encounters ended in a victory for the dynamic ship, 28 per cent in a defeat, and 30 per cent in a tie. This means that 72 per cent of the encounters ended in a situation not disadvantageous to the dynamic ship. The dynamic ship achieved 50 per cent more victories than the static ship. Clearly, on the training set the dynamic ship performs considerably better than the static ship. This supports the statement that the fitness value of 0.579 indicates successful duelling behaviour.

4.1.4 Analysis of Successful Duelling Behaviour

An analysis of the behaviour of the best-performing dynamic ship showed that it exhibited appropriate following behaviour when it overpowered the static ship. In the experiment, such following behaviour is never detrimental to the performance. The reason is that the static ship's game AI ensures that, while fleeing, the static ship will only turn around to attack if the dynamic ship's hull strength becomes less than its own. As long as the dynamic ship remains behind the static ship, this will not happen.

While in pursuit, the dynamic ship avoided bumping against the static ship. Avoiding bumping is appropriate behaviour, because bumping would reduce the dynamic ship's speed to zero, while leaving the static ship's speed unaffected. This would give the static ship an opportunity to escape. However, contrary to expectation the dynamic ship did not avoid bumping by reducing its speed when approaching the static ship, but by swerving as much as needed to keep a constant relative distance to the static ship.

The dynamic ship did not try to flee when losing a fight. The probable reason is that for a spaceship to flee, it must turn its back toward the enemy. The fleeing ship then becomes a target that does not have the ability to fight back (since laser guns only fire from the front of the ship). As a result, fleeing ships are almost always destroyed before being able to escape. Attempts to escape seem therefore of little use. From this observation it can be concluded that in the actual game a better balance between the power of the weapons and the versatility of the ships is required to enable effective escaping behaviour.

The purpose of the experiment was to discover possible improvements to the static ship's game AI. I found two such improvements, which are detailed below.

The first possible improvement was suggested by the dynamic ship's ability to exploit a weakness in the static ship's game AI. The weakness spotted was the following. The static ship bases its decision to flee on a comparison between the relative hull strengths. The comparison does not take into account that it is the static ship's initiative (i.e., turn to act) when it makes the decision. If the comparative hull strengths are close to each other, this becomes an important consideration. For instance, if on the initial approach the static ship comes within the dynamic ship's laser-gun range before being able to fire its own laser guns, it will be damaged while the dynamic ship remains undamaged. Regardless of its own laser-gun strength and hull strength, this would cause the static ship's initial reaction to be attempting to flee. Since in most cases it would still be able to fire its laser guns once, this

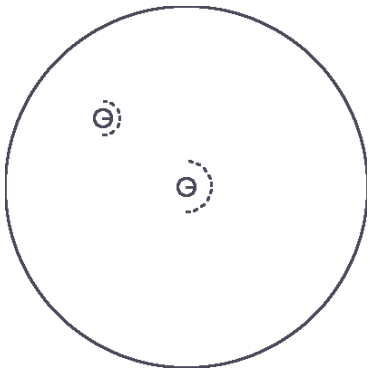

Figure 4.3: The static ship approaches the dynamic ship from behind.

behaviour had little influence when the static ship significantly overpowered the dynamic ship. However, if the strengths of the ships were about equal, we found the dynamic ship to exploit this weakness of the static ship, by attempting to manoeuvre into a position from which it could fire the first shot.[4] Removing this exploit from the static ship's game AI can be considered as a possible improvement.

The second possible improvement was suggested by a surprising manoeuvre of the dynamic ship, that was observed when the static ship started behind the dynamic ship, as illustrated in Figure 4.3. In such cases, the dynamic ship often attempted to increase the distance between the two ships, up until the point where a further increase in separation would imply a tie. At that point, the dynamic ship turned around and either (i) started to attack, or (ii) increased the distance between the two ships again, and attacked after a second turn. Figure 4.4 illustrates this sequence of events. In the figure, the right panel displays a trace of the movements of the dynamic ship up to the moment that it fires its first shot. The static ship is overpowered (its hull strength is very low compared to the hull strength of the dynamic ship, as can be observed at the top of the display) and tries to flee, but the dynamic ship follows, as shown in the left panel. An explanation for the success of the observed behaviour is that, if the distance between the two ships is maximal, the dynamic ship will have a maximal amount of time to turn around and face the static ship before the static ship can fire its laser guns. Since facing the opponent is required to counter-attack, the observed behaviour is beneficial to the dynamic ship's tactics. Below this behaviour is reformulated as a possible improvement of the static ship's game AI.

[4] It is noteworthy that in many commercial turn-based games similar shortcomings in the game AI can be observed. For instance, in many games it is a good tactic for the player to pass game turns until the enemy has approached to a certain distance, so that the player can initiate the first attack. Game designers will seldom let game-playing agents employ such a tactic, because it could lead to a stalemate, where both the player and the computer refuse to move, since whoever makes the first move is at a disadvantage. Similarities with trench warfare are striking.

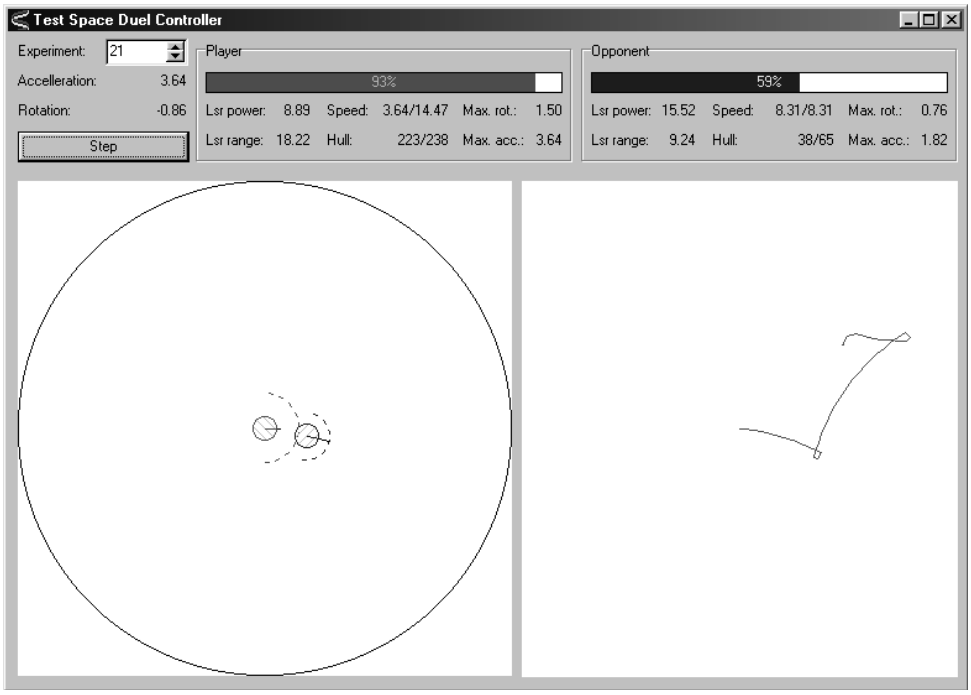

Figure 4.4: The dynamic ship evades the static ship before it attacks.

4.1.5 Deriving Duelling Improvements

The two possible improvements derived from the analysis of the most successful dynamic ship (4.1.4), resulted in two possible changes to the static ship's game AI. The changes are the following.

Fleeing change: Before comparing the hull strength ratios of the two ships, the static ship assumes that it is able to shoot the dynamic ship once more before evaluating the ratios. This change effectively removes the possibility for the dynamic ship to trick the static ship into attempting to flee, when the dynamic ship is able to strike first.

Aft-attack change: When attacked from behind it may be beneficial for the static ship to attempt to increase the distance between the two ships before turning around. This was implemented as follows. First, three conditions are checked, namely (i) whether the dynamic ship is behind the static ship, (ii) whether the static ship is undamaged, and (iii) whether the distance between the ships is in the range $[75, 150]$ (180 being the distance beyond which a fight ends in a tie). If all three conditions are true, then the static ship does not rotate, but simply increases its speed to maximum, in order to increase the distance between

AI	AI-0	AI-1	AI-2	AI-3	
AI-0	0.499 (15/16)	0.481 (15/18)	0.504 (13/15)	0.505 (15/16)	0.497
AI-1	0.525 (18/17)	0.491 (16/17)	0.500 (13/17)	0.504 (15/17)	0.505
AI-2	0.501 (13/14)	0.485 (13/15)	0.494 (10/13)	0.489 (11/13)	0.492
AI-3	0.507 (14/14)	0.487 (13/14)	0.497 (10/13)	0.492 (11/13)	0.496
	0.508	0.486	0.499	0.498	Avg.

Table 4.2: Comparison of four game-AI variations.

the two ships. If the distance becomes larger than 150, it is considered to be sufficiently large to let the static ship turn around safely. If the distance is smaller than 75, the static ship is assumed to be unable to outrun the dynamic ship, so it always turns towards the dynamic ship.

With these two possible changes, four variations of the static ship's game AI can be defined. These are the following. 'AI-0' is the unchanged, original game AI. 'AI-1' is the original game AI, enhanced with the fleeing change. 'AI-2' is the original game AI, enhanced with the aft-attack change. 'AI-3' is the original game AI, enhanced with both the fleeing change and the aft-attack change.

The relative strengths of these four game-AI variations can be derived by pitting them against each other. The results of the cross-comparison are shown in Table 4.2. The rows and columns represent the game AI variations used for the two ships; the ship represented by a row is allowed to move first. The cells of the table show the resulting fitness of the game AI of the first-moving ship. Next to the fitness, between brackets, the number of wins and losses ('wins/losses') is shown. The right column shows the average fitness over the rows, and the bottom row the average fitness over the columns.

It is clear from Table 4.2 that the four game-AI variations do not greatly differ in strength. This comes as no surprise, because their implementations are very similar. The average fitness is highest for AI-1 (0.505), and the average fitness is lowest when it is calculated against an opponent using AI-1 (0.486). Therefore, AI-1 seems to be the most effective of the four variations. However, the difference between AI-1 and the other three variations is too small to be considered significant.

Two unexpected results can be derived from Table 4.2. The first unexpected result is that the fitness values on the main diagonal deviate from 0.5, despite the fact that the competing variations on the diagonal are equal. The deviation is caused by the turn-based handling of the encounters. Since all values on the diagonal are slightly lower than 0.5, it can be concluded that on the 50 training-set cases the second-moving ship has a small advantage over the first-moving ship. Note that this does not entail that initiative is disadvantageous per se, only that it is disadvantageous in the training set.

The second unexpected result concerns the fitness values and the associated win-loss ratios, which in some cases seem counter-intuitive. For instance, AI-0 for the

AI	Tests	Average	Lowest	Highest	Win/loss	Average on test set	Win/loss on test set
AI-0	8	0.537	0.498	0.576	19/14	0.490	16/19
AI-1	6	0.486	0.471	0.528	9/12	0.434	9/20
AI-2	6	0.547	0.479	0.615	16/8	0.476	10/16
AI-3	7	0.517	0.463	0.570	17/11	0.442	13/19

Table 4.3: Results of testing offline evolutionary game AI against four game-AI variations.

first-moving ship, pitted against AI-2 for the second-moving ship, has a fitness value of 0.504. This value, which is slightly greater than 0.5, indicates that AI-0 performs better than AI-2. However, this is combined with 13 wins against 15 losses. Despite the higher fitness value, AI-0 appears weaker than AI-2 in terms of number of wins. The explanation is that the fitness is not based on the number of wins and losses, but on the change of the relative hull strengths during a fight. A fast win might yield a higher fitness than a slow win. As a result, in the fitness rating a few fast wins can compensate for a few extra (slow) losses.

4.1.6 Validating Duelling Improvements

To validate the improvements to the static ship's game AI, the experiment detailed in Subsection 4.1.2 was repeated with three changes: (i) for the static ship I tested all four variations of the game AI defined in Subsection 4.1.5, (ii) because preliminary tests revealed that a feed-forward controller with 5 nodes in each layer was not powerful enough to oppose the new versions of the static ship, for the neural-network controller only a feed-forward controller with two 10-node hidden layers was used, and (iii) the best results achieved on the training set were re-evaluated on five test sets, each consisting of 50 novel encounters.

Table 4.3 shows the results of the validation experiment. From left to right, the eight columns represent: (i) the game AI of the static ship, (ii) the number of experiments performed against this game AI, (iii) the average fitness of the dynamic ship, (iv) the lowest fitness value, (v) the highest fitness value, (vi) the number of wins and losses of the dynamic ship with the highest fitness value, (vii) the average fitness of the best dynamic ship re-evaluated on five test sets, and (viii) the average number of wins and losses for the re-evaluation.

Clearly, on the training set the dynamic ship outperforms three out of four game-AI variations. Only the static ship using AI-1 (which implements the fleeing change) outperforms the dynamic ship. Against AI-1, the dynamic ship has an average fitness lower than 0.5, and even the dynamic ship with the highest fitness value against AI-1 loses more often than the static ship. It is also clear that AI-2 (the game-AI variation that implements the aft-attack change) does not increase the effectiveness of the static ship. AI-2 performs even worse than the original (unchanged) AI-0.

The results of the best dynamic ships on the test sets show that the average fitness drops considerably from its original value. This indicates that, unsurprisingly, the dynamic ship is focused too much on the encounters comprising the training set, i.e., it is overfitting the training set. Interestingly, both the fitness and the win-loss ratio decrease to a larger extent for AI 2 and AI 3 (the game-AI variations that both contain the aft-attack change) than for AI-0 and AI-1. Therefore, overfitting seems to be a more severe problem when trained on AI-2 and AI-3, than when trained on AI-0 and AI-1. Moreover, the dynamic ships evolved against AI-0 and AI-2 (the two game-AI variations that do *not* implement the fleeing change) end up with a significantly higher average fitness on the test sets than the other two game-AI variations. This means that for the dynamic ship it is easier to deal with a game-AI variation that does not implement the fleeing change, than with one that does. Therefore, the conclusion is warranted that implementation of the fleeing change improves the effectiveness of the static ship's game AI.

4.1.7 Discussion of the Duelling Experiments

While implementation of the fleeing change clearly improves the behaviour of the static ship, implementation of the aft-attack change seems to weaken it somewhat. This does not mean that the aft-attack change should not be implemented in a published game. In a game such as PICOVERSE there should be several different game-AI variations available to computer-controlled agents. They must vary in strength and be applicable in various situations. The aft-attack change may be more effective when the situations in which it is a sound tactic can be successfully identified. In addition, allowing some (but not all) agents to use this tactic introduces heterogeneity which makes opponent behaviour less predictable, and thus more entertaining.

In Table 4.2 a discrepancy between the fitness results and the ratio of wins and losses can be observed. Since in terms of game-play experience the win-loss ratio is a more important measure for success than the change in hull strength, the fitness function used is probably not the most suitable for these experiments. In itself, the win-loss ratio is not a good alternative for a fitness measure, because it does not reward small favourable changes in the behaviour of the dynamic ship. However, extending the fitness function with penalties for losing a duel and with extra rewards for winning a duel may improve the correspondence between the fitness rating and the win-loss ratio.

The fact that the results of the re-evaluation of the dynamic ships on the test sets differed considerably from the results on the training set, indicates that the dynamic ship did not generalise to novel situations. A larger training set would probably yield a more general controller, at the cost of a considerably increased computation time. However, in this particular research domain the lack of the ability to generalise is not a problem, as long as existing exploits in the game AI are discovered. The goal of the present experiments is *not* to generate good game AI, but to discover exploits and new tactics.[5] Offline evolutionary game AI managed to achieve that goal.

[5]Of course, that does not mean game AI researchers and developers are not interested in using offline learning to create generalised game AI. Such offline learning will be discussed in Chapter 6.

Chan *et al.* (2004) investigated the evolution of action sequences for FIFA-99. As Spronck *et al.* (2002) concluded, they, too, found that offline evolutionary game AI can be used to detect exploits and discover new tactics. However, instead of a neural network to implement adaptive game AI, they used a Markov Decision Process (MDP), which is arguably a better choice in this respect. Usually, game AI needs to couple environmental circumstances to specific actions for an agent to undertake. The game AI should reflect the human thought process, which game developers aspire to imitate in agents. For this, scripts (which are preferred by most game developers), finite-state machines, and MDPs may be suitable choices, but a neural network is not. Neural networks are suitable to emulate non-linear functions, not production rules. An approach to offline evolutionary learning based on directly evolving scripted AI will be used in Chapter 6.

4.2 Online Evolutionary Game AI

Online evolutionary game AI controls agents that are in competition with human players. It has two applications: (i) to resolve weaknesses in the game AI when they are exploited by the human player (self-correction), and (ii) to create new tactics in response to tactics employed by the human player (creativity). For online evolutionary game AI to be applicable in practice, it must meet the computational requirements of (i) speed, (ii) effectiveness, (iii) robustness, and (iv) efficiency (2.3.4). In general, evolutionary algorithms are computationally intensive (i.e., they are not fast), generate noisy results (i.e., they are not effective), and require numerous experiments (i.e., they are not efficient). Furthermore, in an environment with inherent randomness they can be made robust, but only at the cost of speed and efficiency, which for online learning cannot be spared. These characteristics indicate that it is quite a challenge to implement online evolutionary game AI successfully.

To investigate the potential of online evolutionary game AI, the Team-oriented Evolutionary Adaptability Mechanism (TEAM) was designed. TEAM applies online evolution to game AI that controls a team of agents, that play 'capture-the-flag' in the action game Quake III Arena (henceforth referred to as Quake).[6] This section describes capture-the-flag in Quake (4.2.1), the design of online evolutionary game AI that plays capture-the-flag (4.2.2), the experimental procedure used to test the design (4.2.3), the results of an experiment in which team game AI was evolved (4.2.4), and a discussion of the results (4.2.5).

4.2.1 Capture-the-Flag in Quake

Quake is a '3D shooter' (2.2.2). It has been used by several researchers in their research, because it is popular, state of the art, and highly adaptable (Laird, 2001; Van Waveren and Rothkrantz, 2002). In Quake, a human player controls an agent in a real-time 3D virtual world, called a 'map'. In regular Quake game-play, a

[6] This experiment was performed by Bakkes (2003), in collaboration with and under supervision of the author.

Figure 4.5: QUAKE III ARENA in capture-the-flag game-play mode. A shot is fired at an agent that carries the flag.

player's objective is to eliminate opponent agents. The opponent agents are either controlled by other human players, or by the computer. The map provides agents with objects that can be used to achieve their goals, such as weapons and armour. An eliminated agent is not removed from the game, but 'respawns' at a designated location on the map (Van Waveren and Rothkrantz, 2002).

Capture-the-flag is a team-oriented game-play mode for QUAKE. In capture-the-flag each agent belongs to one of two opposing teams. Each team has a base on the map, and an object representing a flag, that is initially located at the team's base. A team's primary goal in capture-the-flag is to capture the opposing team's flag and bring it to its own base, which scores a point. After delivery of the flag, the flag returns immediately to its starting location. The game is won by the team that scores the most points (Van Waveren and Rothkrantz, 2002). Figure 4.5 shows a screenshot of QUAKE during a capture-the-flag game.

In capture-the-flag mode QUAKE contains two different kinds of game AI, namely (i) agent AI, and (ii) team AI. Agent AI is the game AI that is localised within each individual computer-controlled agent, determining the behaviour of the agent, at an

State	Score	Offensive	Defensive	$N = 4$
No flags stolen	winning	$\max(0.4N, 4)$	$\max(0.5N, 5)$	(2,2,0)
No flags stolen	losing	$\max(0.5N, 5)$	$\max(0.4N, 4)$	(2,2,0)
Home flag stolen	winning	$\max(0.7N, 6)$	$\max(0.3N, 3)$	(3,1,0)
Home flag stolen	losing	$\max(0.7N, 7)$	$\max(0.2N, 2)$	(3,1,0)
Opponent flag stolen	winning	$\max(0.3N, 3)$	$\max(0.6N, 6)$	(1,2,1)
Opponent flag stolen	losing	$\max(0.3N, 3)$	$\max(0.6N, 6)$	(1,2,1)
Both flags stolen	winning	$\max(0.5N, 5)$	$\max(0.4N, 4)$	(2,2,0)
Both flags stolen	losing	$\max(0.5N, 5)$	$\max(0.4N, 4)$	(2,2,0)

Table 4.4: Role divisions of the QUAKE static team AI.

operational level of intelligence. Team AI is the game AI that is implemented as a centralised coach for the computer-controlled team, determining the behaviour of the team as a whole, at a tactical level of intelligence. The team AI provides each of the members of a team with behavioural guidelines. The agent AI takes decisions within the boundaries set by the guidelines (Van der Sterren, 2002).

The team AI implemented in QUAKE by the game developers assigns each team member a role, corresponding to the current game state and the current score. Three different roles are defined, namely (i) offensive, (ii) defensive, and (iii) roaming. Four different game states are defined, distinguishing whether or not each of the two flags is located at its base. Two different score situations are defined, namely whether the team is winning or losing. The implementation of a role differs between game states. For instance, when the opposing team's flag is at its base, an agent with an 'offensive' role attempts to capture that flag. When the opposing team's flag is captured, an agent with an 'offensive' role focuses on attacking members of the opposing team.

The QUAKE team AI is static, i.e., the role division and the role assignments are pre-programmed, although different configurations are used for the four different game states and the two different score situations. The calculations for the eight different role divisions are listed in Table 4.4. The five columns of the table represent (i) the game state, (ii) the score situation ('winning' or 'losing'), (iii) the calculation for the number of team members in an offensive role, (iv) the calculation for the number of team members in a defensive role, and (v) the role division for a team with four members (respectively 'offensive','defensive', and 'roaming'). In the calculations, N represents the total number of team members, and the calculation results are rounded to the nearest integer value.

Adaptive team AI has the ability to tune automatically the team behaviour to the tactics of the opposing team. Therefore, enhancing the QUAKE team AI with adaptive capabilities has the potential to improve a team's behaviour. In the present research, online evolutionary learning is used to implement adaptive team AI.

4.2.2 Adaptive Team AI with TEAM

The Team-oriented Evolutionary Adaptability Mechanism (TEAM) is an online evolutionary learning technique designed to adapt the team AI of QUAKE-like games (Bakkes *et al.*, 2004). TEAM is applicable under the condition that the behaviour of a team in a game is defined by a small number of parameters, specified per game state. A specific instance of team behaviour is defined by values for each of the parameters, for each of the states. TEAM is defined as a regular evolutionary algorithm, such as a genetic algorithm, applied to team-behaviour learning, with the following six properties.

State-based evolution: TEAM employs a separate evolutionary process for each state, each with its own population of chromosomes. The idea is that successful behaviour for each of the separate states can be evolved faster than successful behaviour for all states, acknowledging the requirement that online evolutionary game AI must be efficient. The combination of the best solutions for each of the states is considered to be the best solution for the team AI as a whole.

State-based chromosome encoding: TEAM's chromosomes encode the state's parameters, using real values.

State-transition-based fitness function: TEAM uses a fitness function based on state transitions. Beneficial state transitions reward the chromosome that caused the state transition, while detrimental state transitions punish it. Usually, an assessment of whether a state transition is beneficial or detrimental cannot be given immediately after the transition; it must be delayed until the game has been observed for a while.[7]

Fitness propagation: TEAM propagates fitness values from child chromosomes to their parents. This ensures that a parent chromosome with a high fitness value, that mostly produces children with low fitness values, will get a low fitness value over time. The idea is that such a parent probably achieved high fitness due to chance, and not due to the quality of the solution it represents. This acknowledges the requirement that online evolutionary game AI must be robust.

Elitist selection: TEAM always selects the highest-ranking chromosome to use as parent for the evolution process, acknowledging the requirement that online evolutionary game AI must be effective. While in most applications elitist selection is risky when randomness is involved in the fitness calculation (as is generally the case in games), the fitness-propagation mechanism protects the evolution against inferior top-ranking chromosomes.

[7] For instance, if a state transition happens from a state that is neutral for the team to a state that is good for the team, the transition seems beneficial. However, if this is immediately followed by a second transition to a state that is bad for the team, the first transition cannot be considered beneficial, since it may have been the primary cause for the second transition.

Manually-designed initialisation: TEAM's population is initialised with chromosomes that are designed manually. This ensures that the team AI is effective from the outset, acknowledging the requirement that online evolutionary game AI must be effective.

TEAM differs from reinforcement learning, according to the specifications given by Sutton and Barto (1998), for two of its features, namely that (i) TEAM uses a population (admittedly, in a minor role), and (ii) TEAM uses undirected genetic operators to scan the search space, whereas reinforcement learning uses a gradient-based search.

4.2.3 Experimental Procedure

To evaluate the suitability of TEAM for implementing adaptive team AI, it was tested with the capture-the-flag game-play mode in QUAKE III ARENA. Similar to the experimental procedure used for the duelling experiment (4.1), a dynamic team employing TEAM was pitted against a static team. The static team used the default QUAKE team AI, which has the ability to adapt the team behaviour to the current state of the game. Each team consisted of four agents.

The four game states of QUAKE in capture-the-flag mode, with their state transitions, are illustrated in Figure 4.6. Using D and S to denote the dynamic team's flag and the static team's flag respectively, and the subscripts b and s to denote a flag being at its base and a flag being stolen respectively, the states are defined as (D_b, S_b), (D_s, S_b), (D_b, S_s), and (D_s, S_s). Since events in QUAKE are handled sequentially, in theory transitions are impossible between states that are located diagonally opposite each other in Figure 4.6. From the point of view of the dynamic team, state transitions can be beneficial, indicated with a '+', or detrimental, indicated with a '−'. Depending on the circumstances, some transitions can be both. For instance, when a transition $(D_s, x) \rightarrow (D_b, x)$ occurs, the reason is either that the dynamic team intercepted its stolen flag, which is beneficial, or that the static team scored a point, which is detrimental.

The chromosome used to represent each state was kept small, to elicit speedy evolution. It contained only two parameters, namely (i) the ratio of 'offensive' agents r_o, and (ii) the ratio of 'defensive' agents r_d. Both r_o and r_d were defined as real values in the range [0,1]. Translation of a ratio to the number of agents in the corresponding role, was executed by multiplying the ratio with the total number of agents, rounding up for 'offensive' agents, and rounding down for 'defensive' agents. The assignment of selected roles to specific agents was copied from the default QUAKE team AI. Agents that were assigned neither an 'offensive' role, nor a 'defensive' role, were assigned a 'roaming' role.

After each state transition, a new chromosome was generated for the state in which the game then resided. This chromosome was used to determine the team AI. The team's behaviour under guidance of the new team AI was used to determine the chromosome's fitness $F \in [0, 1]$, according to the following equation.

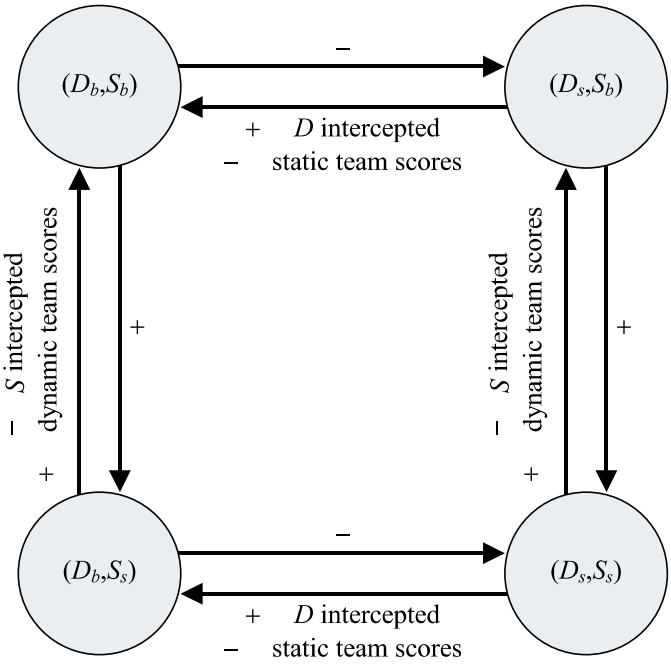

Figure 4.6: State transitions in a capture-the-flag game.

$$F = \sum_{i=T}^{T+d} \frac{F_i}{(T-i)+1} \tag{4.2}$$

In this equation, T is the number of the state transition after which the chromosome was generated, and d is the 'depth' of the calculation, i.e., the number of state transitions that pass before the chromosome's fitness is calculated. In the experiment $d = 2$ was used. The value $F_i \in [0, 1]$ represents the perceived fitness between state transitions i and $i + 1$. F_i is calculated according to the following equation.

$$F_i = \begin{cases} 1 - \min\left(0.1\left(\sqrt{t_i} - \sqrt{t_i/3}\right), 1\right) & \{+ \text{ transition}\} \\ \min\left(0.1\left(\sqrt{t_i} - \sqrt{t_i/3}\right), 1\right) & \{- \text{ transition}\} \end{cases} \tag{4.3}$$

In this equation, t_i is the number of seconds that pass between state transitions i and $i + 1$. The effect of equations 4.2 and 4.3 is that the fitness value awarded to a chromosome is higher when the team AI it represents promotes beneficial state transitions (marked '+' in Figure 4.6), and lower when the team AI it represents promotes detrimental state transitions (marked '−' in Figure 4.6). The longer the resulting game states are maintained, the bigger the effect is.

Recombination operators (genetic operators that use genetic material from multiple parents) often generate children that are radically different from their parents (Davis, 1991), and thus often produce inferior results, which should be avoided on account of the requirement of effectiveness. Therefore, it was decided that only a genetic mutation operator was to be used to generate new chromosomes.

The genetic mutation operator was always applied to the best chromosome in the population. Its effect was scaled in correspondence to the fitness of the parent chromosome it mutated: a parent with a high fitness got a small mutation, while a parent with a low fitness got a large mutation. The mutation was implemented as a biased mutation on one or both genes in the chromosome, while ensuring that the resulting chromosome always represented a legal role division. Newly generated child chromosomes either replaced the bottom-ranking chromosome in the population, or were discarded, if their fitness did not exceed the bottom-ranking chromosome's fitness. With respect to fitness propagation, the fitness calculated for child chromosomes was also factored into the fitness of the parent chromosome.

Since the population's only function is to support the fitness-propagation mechanism, by offering a replacement for the population's top-ranking position in case the current top was removed, a small population size suffices. In the experiment the population size was set to 5. The population was initialised with five copies of a chromosome representing the parameters used by the default QUAKE team AI, to ensure effective behaviour even with the initial dynamic team AI.

4.2.4 Evolving Team AI

The experiment to evaluate the suitability of TEAM for implementing adaptive team AI consisted of fifteen tests. In each test a team using dynamic team AI played QUAKE III ARENA capture-the-flag against a team using static team AI. The game was played on an 'open' map, i.e., a map without walls, allowing the agents an unrestricted view of their environment.

Each test ran for at least six real-time hours, in which between 250 and 600 points were scored. The points scored by each team were tracked, and compared after the tests. The following two measures were defined to rate the success of the dynamic team.

Absolute turning point: The absolute turning point is the number of the last point scored, after which the dynamic team's total score exceeds the static team's total score for the remainder of the test. Figure 4.7 illustrates the absolute turning point with a graph displaying the dynamic team's lead in one of the tests. After point 52 is scored, the dynamic team's score exceeds the static team's score for the remainder of the test. Therefore, in this example the absolute turning point is 52.

Relative turning point: The relative turning point is the number of the last point in the first sliding window of twenty points, in which the dynamic team scored fifteen, and the static team scored five points. At the relative turning point the dynamic team's behaviour is more successful than the static team's behaviour

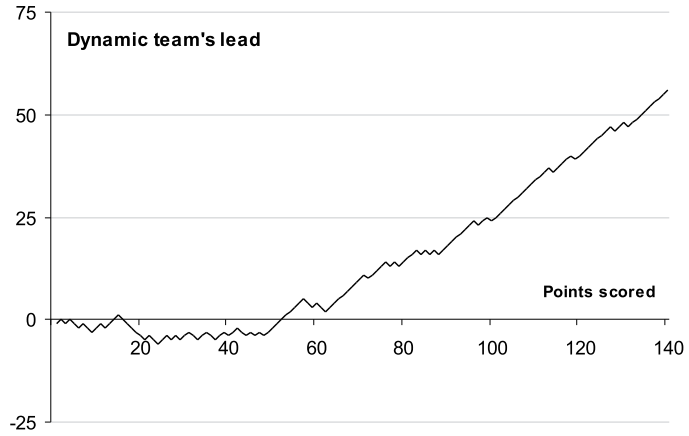

Figure 4.7: A test run with an absolute turning point of 52.

with a reliability $> 97\%$ (Cohen, 1995). Figure 4.8 illustrates the relative turning point with a graph displaying the dynamic team's number of wins in a sliding window of 20 points scored, in the same test used for Figure 4.7. At the scoring of point 57, the dynamic team's score in the window of the last twenty points scored is fifteen for the first time. Therefore, in this example the relative turning point is 57. Note that, due to the window size of 20, the lowest possible value for the relative turning point is 20.

Fifteen tests were performed. In all tests the dynamic team managed to evolve team AI which allowed it to defeat the static team consistently. Table 4.5 provides an

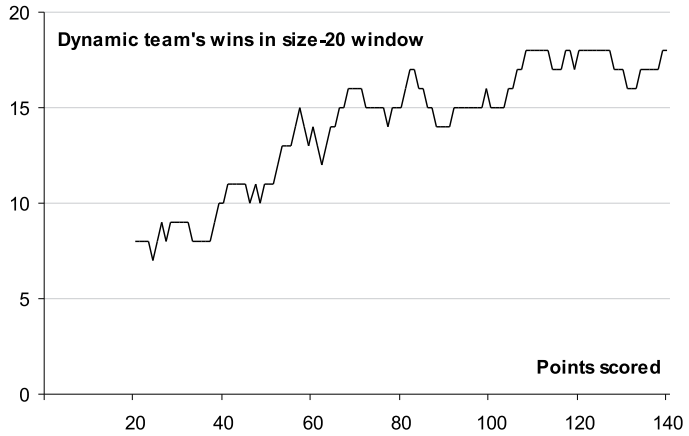

Figure 4.8: A test run with a relative turning point of 57.

	Average	St.dev.	Median	Highest	Lowest
Absolute turning point	108	62.0	99	263	38
Relative turning point	71	44.8	50	158	20

Table 4.5: Results for the team-AI experiment.

overview of the results. From these results it can be concluded that TEAM is capable of successfully adapting team behaviour in QUAKE capture-the-flag. Analysing the behaviour of the evolved team AI, it was observed that the dynamic team used risky, but successful, tactics against the static team. The tactics can best be described as 'rush' tactics, aimed at quickly obtaining offensive field supremacy.[8] The default QUAKE team AI only applies 'moderate' tactics, leaving at least one agent in a 'defensive' role, and is therefore unable to deal effectively with rush tactics.

4.2.5 Discussion of the Team-AI Experiment

In the introduction of Section 4.2, is was indicated that it is hard to create online evolutionary game AI that meets the four computational requirements for online learning in games (detailed in 2.3.4). The four requirements are now discussed for the team-AI experiment.

- *Speed*: The implementation of the dynamic team AI, using a small chromosome and a small population, needed relatively few processing cycles. During the tests, the game-play was never interrupted or slowed down because of the evolutionary process. Therefore, it can be concluded that the dynamic team AI meets the requirement of speed.

- *Effectiveness*: Table 4.5 shows that, on average, the absolute turning point is significantly higher than the relative turning point. This means that, in general, the dynamic team has become the dominant team on the map a considerable period of time before it actually gains the lead in the number of points scored. The reason for the gap between the two turning points is that initially the dynamic team tends to be weaker than the static team. However, it was observed during all fifteen tests that its score never was more than about a dozen points behind the static team's score. In contrast, as soon as the absolute turning point was reached, the dynamic team's lead increased to hundreds of points. Therefore, it can be concluded that the dynamic team AI meets the requirement of effectiveness.

[8] The dynamic team AI assigns all agents an 'offensive' role in the state (D_b, S_b). In translation, this means that in a situation where its own flag is in no immediate danger, and the opponent's flag is not captured, the dynamic team will launch an all-out attack to get the opponent's flag as quickly as possible, which is the first step that needs to be taken to score a point. Rush tactics are often applied in real-time strategy games, which are discussed in Chapter 6.

- *Robustness*: In almost all tests the dynamic team AI did not suffer from the inherent randomness in the QUAKE environment. Only in one of the fifteen tests, the dynamic team AI, after having increased its lead to about 375 points, suddenly seemed to 'forget' the successful tactics it had learned, and started losing again. After its lead had dropped to about 340 points, it recovered. The explanation for this phenomenon is that the dynamic team AI had difficulties dealing with a long run of fitness values that, due to chance, were not representative for the quality of the chromosome they were assigned to. It is possible to protect the dynamic team AI better against such chance runs, by not replacing the team AI after each state transition. Instead, the time gained is used to confirm the assigned fitness values. The drawback is that this will hurt the efficiency of the process. Moreover, statistically it is impossible to rule out such chance runs completely. Taking all these facts into account, it can be concluded that the dynamic team AI is fairly robust.

- *Efficiency*: When in a capture-the-flag game the relative turning point is reached, the dynamic team's superiority is clear. Table 4.5 shows that the average relative turning point is 71, i.e., after the scoring of only 71 points the dynamic team significantly outperforms the static team. A relative turning point of 71 is quite low, considering that, in general, evolutionary algorithms need thousands of trials (or more) to find an acceptable solution. Therefore, at first glance the dynamic team AI seems to be efficient. However, for three reasons we should be cautious in regarding this result too optimistically. The reasons are the following. (i) As the high standard deviation of 44.8 indicates, the relative turning point has a high variance, which is in conflict with the functional requirement of consistency (2.3.4). (ii) With four states and basically only fifteen different allele combinations per chromosome,[9] the search space for team AI covering all four states only contains $15^4 = 50625$ different solutions, and thus is very small. (iii) The dynamic team started with tactics equal to the already effective tactics used by the static team. On average, the dynamic team needed about two hours of real-time play to turn the effective initial tactics into superior tactics. In general, QUAKE capture-the-flag matches do not last that long. Taking the three reasons into account, it can be concluded that the dynamic team AI is moderately efficient, provided the search space is small.

TEAM can be applied in practical situations, because it does not slow down game-play, its tactics do not degrade, and it is fairly robust. While it is lacking in efficiency, in capture-the-flag matches that run for long periods of time, it may be expected that TEAM will discover successful tactics, under the provision that the search space is small.

[9] Let $N_o \in \mathbb{N}$ be the number of agents that gets an 'offensive' role, $N_d \in \mathbb{N}$ be the number of agents that gets a 'defensive' role, and $N \in \mathbb{N}$ be the total number of agents in a team. Then it holds that $N_o + N_d \leq N$. With $N = 4$ agents in a team, as used in the team-AI experiment, only fifteen different role divisions are possible.

4.3 Discussion of Evolutionary Game AI

Offline evolutionary game AI achieved good results in exploiting weaknesses in game AI, and in discovering new tactics, in the duelling-spaceships environment described in Section 4.1. This is of no surprise, since the only requirement for use of evolutionary learning is that an adequate fitness function can be designed (Goldberg, 1989). A fitness function for the evolution of tactics in a game may be designed by taking into account the speed by which an encounter is played out, and the effectiveness by which agents defend themselves and attack the human player. In general, games provide such information. Thus, it may be concluded that evolutionary learning can be used to detect exploits in game AI, and to design new tactics for game AI.

In the duelling-spaceships experiment, a neural network was used to implement the game AI. It was argued that a neural network is not a suitable architecture to store game AI, because it cannot create the equivalent of scripts consisting of production rules. In Chapter 6, where offline evolutionary game AI will be applied to a different problem, an alternative learning structure will be used, specifically designed to evolve production rules. However, the same overall design as used in the present chapter will be used, namely evolving strong tactics by pitting offline evolutionary game AI against strong static game AI.

In the QUAKE capture-the-flag experiment described in Section 4.2, online evolutionary game AI achieved good results in improving tactics against a specific opponent during QUAKE game-play. The opponent was the standard opponent implemented by the QUAKE developers, with the ability to switch between different configurations in response to changing circumstances. Despite the good results, the learning mechanism was shown to be only moderately efficient.

Laird (2000) is skeptical about the possibilities offered by online evolutionary game AI. He states that, while neural networks and evolutionary algorithms may be applied to tune parameters, they are "grossly inadequate when it comes to creating synthetic characters with complex behaviours automatically from scratch". In contrast, the results achieved with dynamic team AI in QUAKE show that it is certainly possible to use online evolutionary algorithms for game AI design. A similar discovery, using online evolutionary learning to evolve agent AI, was made by Demasi and Cruz (2002).

However, the team AI designed for QUAKE capture-the-flag, and the agent AI designed by Demasi and Cruz (2002), are both simple, controlled by just a few parameters. Regarding the '*complex* behaviours' referred to in Laird's sentiment, it is highly doubtful whether an evolutionary approach can generate those in an efficient manner. It is likely that the search for complex behaviour takes place in a large search space. In general, the larger the search space, the less efficient an evolutionary algorithm (or, indeed, any other search algorithm) will be (Russell and Norvig, 2003). When online evolutionary game AI is no longer efficient, its practical use is negligible.

It may be concluded that evolutionary game AI is suitable for the offline adaptation of game AI, and for the online adaptation of game AI for *simple* behaviour. However, for lack of efficiency it is not the right approach for the online adaptation

of game AI for *complex* behaviour. A different approach to online adaptation of game AI, targeted at the adaptation of complex behaviour, will be introduced in Chapter 5.

4.4 Chapter Summary

In this chapter both offline and online evolutionary game AI were investigated. Offline evolutionary game AI was shown to be able to exploit weaknesses in game AI, and to discover new tactics, when pitted against strong static game AI. Online evolutionary game AI was shown to be able to improve tactics against a specific opponent during game-play. However, the success of online evolutionary game AI depended on the potential solutions residing in a small search space. In general, when evolving game AI that is complex, online evolutionary game AI will not be sufficiently efficient. Efficiency is a requirement to apply online adaptation of game AI in practice. Therefore, to adapt complex game AI, a different approach needs to be used.

Chapter 5

Dynamic Scripting

When error is corrected whenever it is recognised as such,
the path of error is the path of truth.
— Hans Reichenbach (1891–1953).

In Chapter 4 it was shown that online evolutionary game AI fails to meet one of the computational requirements for online-learning, namely the requirement of efficiency (2.3.4). The present chapter[1] discusses online learning of game AI using a novel technique called 'dynamic scripting'. Dynamic scripting has been designed to meet all four computational online-learning requirements. With a few enhancements, it is also able to meet all four functional requirements. Section 5.1 introduces the dynamic-scripting technique. Experiments performed for evaluating the adaptive performance of dynamic scripting are described in Sections 5.2 to 5.5. Section 5.2 describes the experimental procedure, and investigates the performance of dynamic scripting in a simulated CRPG. Section 5.3 investigates enhancements to dynamic scripting to reduce the number of exceptionally long adaptation runs. Section 5.4 investigates enhancements to dynamic scripting to allow scaling of the difficulty level of the game AI to the experience level of the human player. In Section 5.5, the results achieved in the simulated CRPG are validated in an actual state-of-the-art CRPG. A summary of the chapter is provided in Section 5.6.

5.1 Dynamic-Scripting Technique

This section describes the dynamic-scripting technique (5.1.1), provides pseudo-code for two of its main process (5.1.2), and explains to what extent it meets the computational and functional requirements for online learning of game AI (5.1.3).

[1] This chapter is based on three papers by Spronck, Sprinkhuizen-Kuyper, and Postma (2004a; 2004b; 2004c).

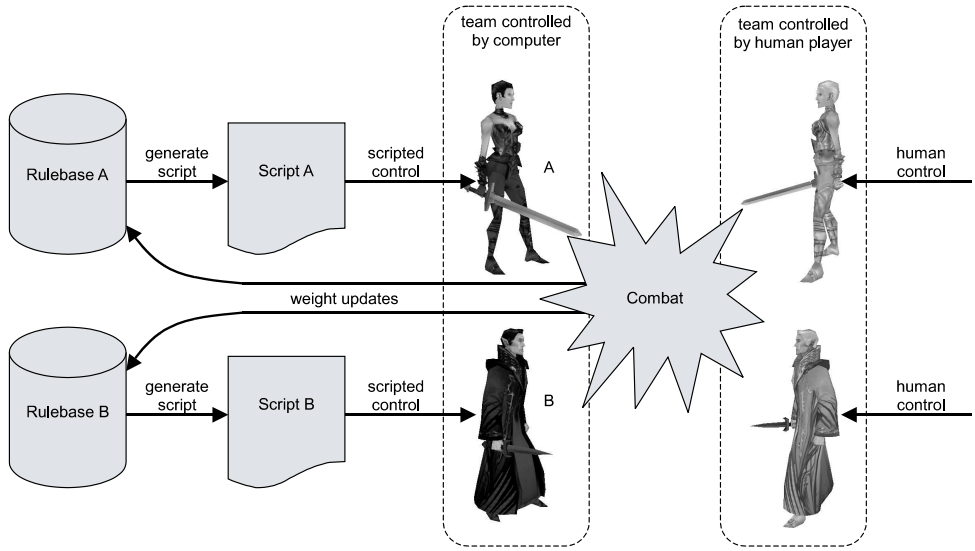

Figure 5.1: Dynamic scripting.

5.1.1 Description of Dynamic Scripting

Dynamic scripting is an online machine-learning technique for game AI, that can be characterised as a stochastic optimisation technique. Dynamic scripting maintains several rulebases, one for each agent class in the game. Every time a new instance of an agent is generated, the rulebases are used to create a new script that controls the agent's behaviour. The rules that comprise a script controlling a particular agent are extracted from the rulebase associated with the agent's class. The probability that a rule is selected for a script is influenced by a weight value that is attached to each rule. Adaptation of the rulebase proceeds by changing the weight values to reflect the success or failure rate of the corresponding rules in scripts. The weight changes are determined by a weight-update function.

The dynamic-scripting technique is illustrated in Figure 5.1 in the context of a commercial game. In the figure, the team dressed in grey is controlled by a human player, while the computer controls the team dressed in black. The rulebase associated with each computer-controlled agent (named 'A' and 'B' in Figure 5.1) contains manually-designed rules derived from domain-specific knowledge. It is imperative that the majority of the rules in the rulebase define effective, or at least sensible, agent behaviour.

At the start of an encounter (i.e., a fight between two opposing teams), a new script is generated for each computer-controlled agent, by randomly selecting a specific number of rules from its associated rulebase. There is a linear relationship between the probability that a rule is selected and its associated weight. The order in which the rules are placed in the script depends on the application domain. A

priority mechanism can be used to let certain rules take precedence over other rules. Such a priority mechanism is only required if a general ordering of rules and actions is prescribed by the domain knowledge. More specific action groupings, such as two actions which must always be executed in a specific order, should be combined in one rule.

The learning mechanism in the dynamic-scripting technique is inspired by reinforcement learning techniques (Sutton and Barto, 1998; Russell and Norvig, 2003). 'Regular' reinforcement learning techniques, such as TD-learning, in general need large amounts of trials, and thus do not meet the requirement of efficiency (Manslow, 2002; Madeira *et al.*, 2004). Reinforcement learning may be suitable for online learning of game AI when the trials occur in a short time-span. Such may be the case on an operational level of intelligence, as in, for instance, the work by Graepel *et al.* (2004), where fight movements in a fighting game are learned. However, for the learning on a tactical or strategic level of intelligence, a trial consists of observing the performance of a tactic over a fairly long period of time. Therefore, for the online learning of tactics in a game, reinforcement learning will take too long to be particularly suitable. In contrast, dynamic scripting has been designed to learn from a few trails only.

In the dynamic-scripting approach, learning proceeds as follows. Upon completion of an encounter (combat), the weights of the rules employed during the encounter are adapted depending on their contribution to the outcome. Rules that lead to success are rewarded with a weight increase, whereas rules that lead to failure are punished with a weight decrease. The increment or decrement of each weight is compensated for by decreasing or increasing all remaining weights as to keep the weight total constant.

Dynamic scripting can be applied to any form of game AI that meets three requirements: (i) the game AI can be scripted, (ii) domain knowledge on the characteristics of a successful script can be collected, and (iii) an evaluation function can be designed to assess the success of the script. Note that the maximum playing strength game AI can achieve using dynamic scripting depends on the quality of the domain knowledge used to create the rules in the rulebase. In the present chapter, it is assumed that the game developer provides high-quality domain knowledge. In Chapter 6, I discuss the automatic generation of high-quality domain knowledge.

5.1.2 Dynamic Scripting Code

The two central processes of the dynamic-scripting technique are script generation and weight adjustment, which are specified in pseudo-code in this subsection. In the code, the rulebase is represented by an array of *rule* objects. Each *rule* object has three attributes, namely (i) *weight*, which stores the rule's weight as an integer value, (ii) *line*, which stores the rule's actual text to add to the script when the rule is selected, and (iii) *activated*, which is a boolean that indicates whether the rule was activated during script execution.

Algorithm 1 presents the script generation algorithm. In the algorithm, the function 'InsertInScript' add a line to the script. If the line is already in the script,

Algorithm 1 Script Generation

```
 1: ClearScript()
 2: sumweights = 0
 3: for i = 0 to rulecount − 1 do
 4:    sumweights = sumweights + rule[i].weight
 5: end for
 6: for i = 0 to scriptsize − 1 do
 7:    try = 0
 8:    lineadded = false
 9:    while try < maxtries and not lineadded do
10:       j = 0
11:       sum = 0
12:       selected = −1
13:       fraction = random(sumweights)
14:       while selected < 0 do
15:          sum = sum + rule[j].weight
16:          if sum > fraction then
17:             selected = j
18:          else
19:             j = j + 1
20:          end if
21:       end while
22:       lineadded = InsertInScript(rule[selected].line)
23:       try = try + 1
24:    end while
25: end for
26: FinishScript()
```

the function has no effect and returns 'false'. Otherwise, the line is inserted and the function returns 'true'. The algorithm aims to put *scriptsize* lines in the script, but may end up with less lines if it needs more than *maxtries* trials to find a new line. The function 'FinishScript' appends one or more lines to the script, to ensure that the script will always find an action to execute. For computational speed, all numbers in the algorithm are integer values.

Algorithm 2 presents the weight adjustment algorithm. The function 'CalculateAdjustment' calculates the reward or penalty each of the activated rules receives. The parameter *Fitness* is a measure of the performance of the script during the encounter. For computational speed, all numbers in the algorithm are integer values, except for the value of *Fitness*, which is a real value.

Note that in Algorithm 1 the calculation of *sumweights* in lines 3 to 5 should always lead to the same result, namely the sum of all the initial rule weights. However, the short calculation that is used to determine the value of *sumweights* ensures that the algorithm will succeed even if Algorithm 2 does not divide the value of *remainder* completely (to avoid using too many processing cycles).

Algorithm 2 Weight Adjustment

```
1: active = 0
2: for i = 0 to rulecount − 1 do
3:    if rule[i].activated then
4:       active = active + 1
5:    end if
6: end for
7: if active <= 0 or active >= rulecount then
8:    return {No updates are needed.}
9: end if
10: nonactive = rulecount − active
11: adjustment = CalculateAdjustment(Fitness)
12: compensation = −round(active ∗ adjustment/nonactive)
13: remainder = −active ∗ adjustment − nonactive ∗ compensation
14: {Awarding rewards and penalties:}
15: for i = 0 to rulecount − 1 do
16:    if rule[i].activated then
17:       rule[i].weight = rule[i].weight + adjustment
18:    else
19:       rule[i].weight = rule[i].weight + compensation
20:    end if
21:    if rule[i].weight < minweight then
22:       remainder = remainder + (rule[i].weight − minweight)
23:       rule[i].weight = minweight
24:    else if rule[i].weight > maxweight then
25:       remainder = remainder + (rule[i].weight − maxweight)
26:       rule[i].weight = maxweight
27:    end if
28: end for
29: {Division of remainder:}
30: i = 0
31: while remainder > 0 do
32:    if rule[i].weight <= maxweight − 1 then
33:       rule[i].weight = rule[i].weight + 1
34:       remainder = remainder − 1
35:    end if
36:    i = (i + 1) mod rulecount
37: end while
38: while remainder < 0 do
39:    if rule[i].weight >= minweight + 1 then
40:       rule[i].weight = rule[i].weight − 1
41:       remainder = remainder + 1
42:    end if
43:    i = (i + 1) mod rulecount
44: end while
```

5.1.3 Dynamic Scripting and Learning Requirements

Dynamic scripting meets five of the eight computational and functional requirements (2.3.4) by design, as follows.

- *Speed* (computational): Dynamic scripting is computationally fast, because it only requires the extraction of rules from a rulebase and the updating of weights once per encounter.
- *Effectiveness* (computational): Dynamic scripting is effective, because all rules in the rulebase are based on domain knowledge. Therefore, every action which an agent executes through a script that contains these rules, is an action that is at least reasonably effective (although it may be inappropriate for certain situations). Note that if the game developers make a mistake and include an inferior rule in the rulebase, the dynamic-scripting technique will quickly assign this rule a low weight value. Therefore, the requirement of effectiveness is met even if the rulebase contains a few inferior rules.
- *Robustness* (computational): Dynamic scripting is robust, because rules are not removed immediately when punished. Instead, they get selected less often. Their selection rate will automatically increase again, either when they are included in a script that achieves good results, or when other rules are punished.
- *Clarity* (functional): Dynamic scripting generates scripts, which can be easily understood by game developers.
- *Variety* (functional): Dynamic scripting generates a new script for every agent, and thus provides a high variety in behaviour.

The remaining three requirements, namely the computational requirement of efficiency and the functional requirements of consistency and scalability, are not met by design. The dynamic-scripting technique is believed to meet the requirement of efficiency, because with appropriate weight-updating parameters it can adapt after a few trials only. This is investigated empirically in Section 5.2. Enhancements to the dynamic-scripting technique that make it meet the requirements of consistency and scalability are investigated in Sections 5.3 and 5.4, respectively.

5.2 Efficiency Validation

Since the dynamic-scripting technique is designed to be used against human players, ideally an empirical evaluation of the technique is derived from an analysis of games it plays against humans. However, due to the huge number of tests that must be performed, such an evaluation is not feasible within a reasonable amount of time (Madeira *et al.*, 2004). Therefore, I decided to evaluate the dynamic-scripting technique by its ability to discover scripts capable of defeating strong, but static, tactics. Translated to a game played against human players, the evaluation tests

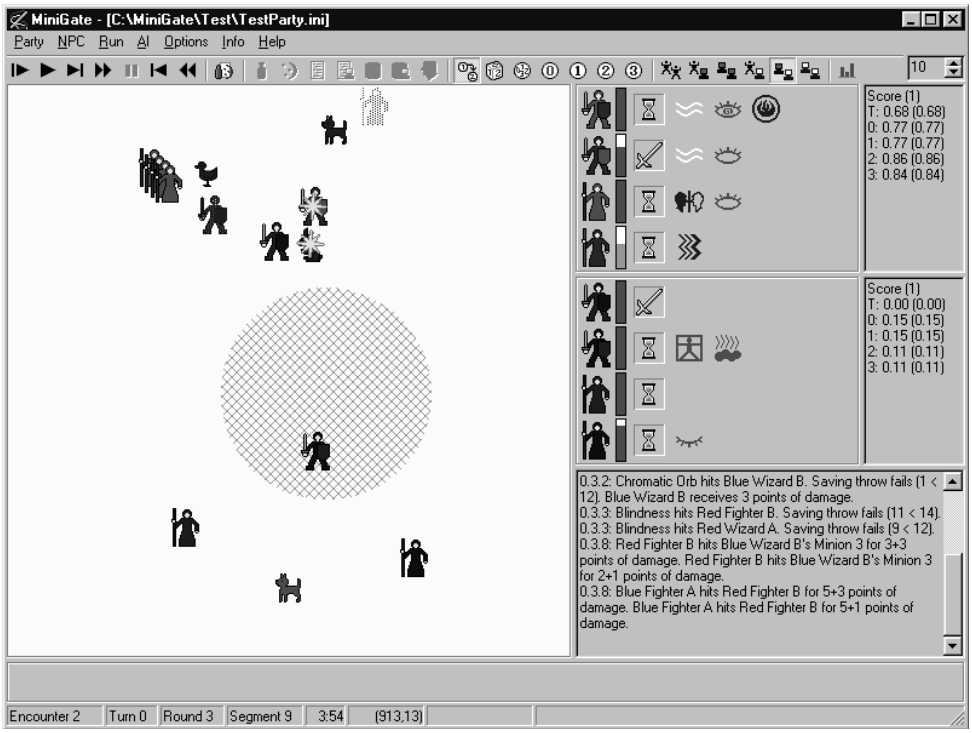

Figure 5.2: The CRPG simulation.

the ability of the dynamic-scripting technique to force the human player to seek continuously new tactics, because the game AI will automatically adapt to deal with tactics that are used often. The evaluation was performed in a simulated CRPG. This section describes the simulation environment (5.2.1), the scripts and rulebases (5.2.2), the weight-update function (5.2.3), the tactics against which the dynamic-scripting technique is tested (5.2.4), the measures used to evaluate the results (5.2.5), and the achieved experimental results (5.2.6).

5.2.1 Simulation Environment

The CRPG simulation used to evaluate dynamic scripting is illustrated in Figure 5.2. It is modelled after the popular Baldur's Gate games. These games (along with a few others) contain the most complex and extensive game-play system found in modern CRPGs, closely resembling classic non-computer roleplaying games (Cook, Tweet, and Williams, 2000). The simulation entails an encounter between two teams of similar composition. The 'dynamic team' is controlled by dynamic scripting. The 'static team' is controlled by unchanging scripts, that represent strong tactics. Each team consists of four agents, namely two 'fighters' and two 'wizards' of equal

'experience level'. The armament and weaponry of the teams is static, and each agent is allowed to select two (out of three possible) magic potions. In addition, the wizards are allowed to memorise seven (out of 21 possible) magic spells. The spells incorporated in the simulation are of varying types, amongst which damaging spells, blessings, curses, charms, area-effect spells, and summoning spells.

The simulation is implemented with hard constraints and soft constraints. Hard constraints are constraints that are submitted by the games rules, e.g., a hard constraint on spells is that they can only be used when they are memorised, and a hard constraint on agents is that they can only execute an action when they are not incapacitated. Soft constraints are constraints that follow as logical consequences from the rules, e.g., a soft constraint on a healing potion is that only an agent that has been damaged should drink it. Both hard and soft constraints are taken into account when a script is executed, e.g., agents will not drink a healing potion when they are incapacitated or undamaged.

In the simulation, the practical issue of choosing spells and potions for agents is solved by making the choice depend on the (generated) scripts, as follows. Before the encounter starts, the scripts are scanned to find rules containing actions that refer to drinking potions or casting spells. When such a rule is found, a potion or spell that can be used in that action is selected. If the agent controlled by the script is allowed to possess the potion or spell, it is added to the agent's inventory.

More details on the CRPG simulation environment can be found in Appendix A, Section A.1.

5.2.2 Scripts and Rulebases

The scripting language was designed to emulate the power and versatility of the scripts used in the BALDUR'S GATE games. The scripting language is explained in detail in Appendix A, Section A.2.

Rules in the scripts are executed in sequential order. For each rule the condition (if present) is checked. If the condition is fulfilled (or absent), the action is executed if it obeys all relevant hard and soft constraints. If no action is selected when the final rule is checked, the default action 'pass' is used.

When dynamic scripting generates a new script, the rule order in the script is determined by a manually-assigned priority value. Rules with a higher priority take precedence over rules with a lower priority. In case of equal priority, the rules with higher weights take precedence. For rules with equal priorities and equal weights, the order is determined randomly.

The selection of script sizes was motivated by the following two considerations, namely that (i) a fighter has less action choices than a wizard, thus a fighter's script can be shorter than a wizard's script, and (ii) a typical fight will last five to ten rounds, thus a maximum of ten rules in a script seems sufficient. Therefore, the size of the script for a fighter was set to five rules, which were selected out of a rulebase containing twenty rules. For a wizard, the script size was set to ten rules, which were selected out of a rulebase containing fifty rules. At the end of each script, default rules were attached, to ensure the execution of an action in case none of the

rules extracted from the rulebase could be activated. The rulebases used are listed in Appendix A, Section A.3.

5.2.3 Weight-Update Function

The weight-update function is based on two so-called 'fitness functions', namely (i) a team-fitness function $F(g)$ (where g refers to the team), and (ii) an agent-fitness function $F(a, g)$ (where a refers to the agent, and g refers to the team to which the agent belongs). The fitness functions have been designed with the aim to assign high fitness to behaviour that manages to defeat the opposing team, or that at least manages to put up a good fight.

Both fitness functions yield a value in the range $[0, 1]$. The fitness values are calculated at time $t = T$, where T is the time step at which all agents in one of the teams are 'defeated', i.e., have their health reduced to zero or less. A team of which all agents are defeated, has lost the fight. A team that has at least one agent 'surviving', has won the fight. At rare occasions both teams may lose at the same time.

The team-fitness function is defined as follows.

$$F(g) = \sum_{c \in g} \begin{cases} 0 & \{g \text{ lost}\} \\ \dfrac{1}{2N_g}\left(1 + \dfrac{h_T(c)}{h_0(c)}\right) & \{g \text{ won}\} \end{cases} \tag{5.1}$$

In this equation, g refers to a team, c refers to an agent, $N_g \in \mathbb{N}$ is the total number of agents in team g, and $h_t(c) \in \mathbb{N}$ is the health of agent c at time t. According the equation, a 'losing' team has a fitness of zero, while the 'winning' team has a fitness exceeding 0.5.

The agent-fitness function is defined as follows.

$$F(a, g) = \frac{1}{10}\Big(3F(g) + 3A(a) + 2B(g) + 2C(g)\Big) \tag{5.2}$$

In this equation, a refers to the agent whose fitness is calculated, and g refers to the team to which agent a belongs. The equation contains four components, namely (i) $F(g)$, the fitness of team g, derived from equation 5.1, (ii) $A(a) \in [0, 1]$, which is a rating of the survival capability of agent a, (iii) $B(g) \in [0, 1]$, which is a measure of health of all agents in team g, and (iv) $C(g) \in [0, 1]$, which is a measure of damage done to all agents in the team opposing g. The weight of the contribution of each of the four components to the final outcome was determined arbitrarily, taking into account the consideration that agents should give high rewards to a team victory, and to their own survival (expressed by the components $F(g)$ and $A(a)$, respectively). The function assigns smaller rewards to the survival of the agent's comrades, and to the damage inflicted upon the opposing team (expressed by the components $B(g)$ and $C(g)$, respectively). As such the agent-fitness function is a good measure of the success rate of the script that controls the agent.

The components $A(a)$, $B(g)$, and $C(g)$ are defined as follows.

$$A(a) = \frac{1}{3} \begin{cases} \min\left(\dfrac{D(a)}{D_{max}}, 1\right) & \{h_T(a) \leq 0\} \\ 2 + \dfrac{h_T(a)}{h_0(a)} & \{h_T(a) > 0\} \end{cases} \tag{5.3}$$

$$B(g) = \frac{1}{2N_g} \sum_{c \in g} \begin{cases} 0 & \{h_T(c) \leq 0\} \\ 1 + \dfrac{h_T(c)}{h_0(c)} & \{h_T(c) > 0\} \end{cases} \tag{5.4}$$

$$C(g) = \frac{1}{2N_{\neg g}} \sum_{c \notin g} \begin{cases} 1 & \{h_T(c) \leq 0\} \\ 1 - \dfrac{h_T(c)}{h_0(c)} & \{h_T(c) > 0\} \end{cases} \tag{5.5}$$

In equations 5.3 to 5.5, a and g are as in equation 5.2, c, N_g and $h_t(c)$ are as in equation 5.1, $N_{\neg g} \in \mathbb{N}$ is the total number of agents in the team that opposes g, $D(a) \in \mathbb{N}$ is the time of 'death' of agent a, and D_{max} is a constant (D_{max} was set to 100 in the experiments, which equals ten combat rounds, which is longer than most fights last).

The agent fitness is translated into weight adaptations for the rules in the script. Weight values are bounded by a range $[W_{min}, W_{max}]$, with excess rewards being redistributed over all weights. Only the rules in the script that are actually executed during an encounter are rewarded or penalised. A new weight value is calculated as $W + \triangle W$, where W is the original weight value, and the weight adjustment $\triangle W$ is expressed by the following formula.

$$\triangle W = \begin{cases} -\lfloor P_{max} \dfrac{b - F}{b} \rfloor & \{F < b\} \\ \lfloor R_{max} \dfrac{F - b}{1 - b} \rfloor & \{F \geq b\} \end{cases} \tag{5.6}$$

In this equation, $R_{max} \in \mathbb{N}$ and $P_{max} \in \mathbb{N}$ are the maximum reward and maximum penalty respectively, F is the agent fitness, and $b \in \langle 0, 1 \rangle$ is the break-even value. At the break-even point the weights remain unchanged. To keep the sum of all weight values in a rulebase constant, weight changes are executed through a redistribution of all weights in the rulebase. The weight-adjustment formula is visualised later in this chapter, in figure 5.6 (left).

In the efficiency-validation experiment, values for the constants were set as follows. The break-even value b was set to 0.3, since in the simulation this value is between the fitness value that the 'best losing agent' achieves and the fitness value that the 'worst winning agent' achieves (about 0.2 and 0.4, respectively). The initialisation of the rulebase assigned all weights the same weight value, $W_{init} = 100$.

W_{min} was set to zero to allow rules that are punished a lot to be effectively removed from the script-generation process. W_{max} was set to 2000, which is such a high value that it allows weights to grow more or less unrestricted. R_{max} was set to 100 to increase the efficiency of dynamic scripting by allowing large weight increases for agents with a high fitness. P_{max} was set to 30, which is relatively small compared to R_{max}, to protect the rulebase from degradation as soon as a local optimum is found. Intuitively, the argument for the low value of P_{max} seems to be correct, since the penalty is similar to the mutation rate in evolutionary algorithms, which should be small in the neighbourhood of an optimum (Bäck, 1996). However, in Section 5.3 it will be shown that a higher value for the maximum penalty gives a better performance for dynamic scripting.

5.2.4 Tactics

Four different basic tactics and three composite tactics were defined for the static team. The four basic tactics, implemented as a static script for each agent of the static team, are as follows (in these description, an 'enemy' is a member of the dynamic team).

Offensive: The fighters always attack the nearest enemy with a melee weapon, while the wizards use the most damaging offensive spells at the (according to domain knowledge) most susceptible enemies.

Disabling: The fighters start by drinking a potion that protects them from any disabling effect, then attack the nearest enemy with a melee weapon. The wizards use all kinds of spells that incapacitate enemies for several rounds.

Cursing: The fighters always attack the nearest enemy with a melee weapon, while the wizards use all kinds of spells that reduce the enemies' effectiveness, e.g., they try to charm enemies (i.e., turn them into allies), physically weaken enemy fighters, deafen enemy wizards (which causes many of the spells they cast to fail), and summon minions in the middle of the enemy team.

Defensive: The fighters start by drinking a potion that reduces fire damage, after which they attack the closest enemy with a melee weapon. The wizards use all kinds of defensive spells, to deflect harm from themselves and from their comrades, including the summoning of minions.

Details of the basic tactics are listed in Appendix A, Section A.4.

To assess the ability of the dynamic-scripting technique to cope with sudden changes in tactics, the following three composite tactics were defined.

Random team: For each encounter, one of the four basic tactics is selected randomly.

Random agent: For each encounter, each agent randomly selects one of the four basic tactics, independent from the choices of his comrades.

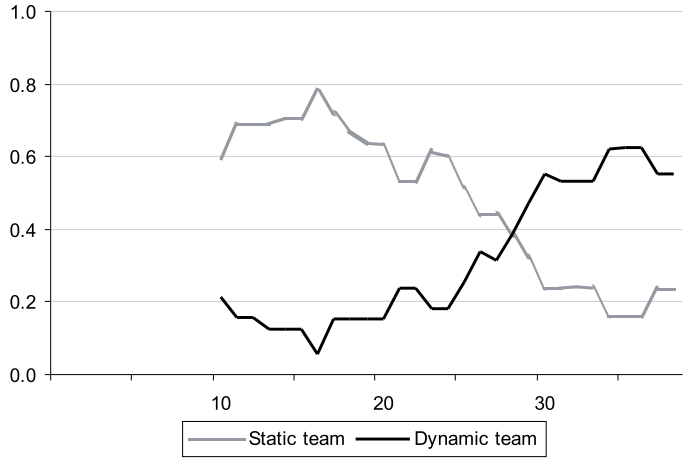

Figure 5.3: Average fitness in size-10 window progression.

Consecutive: The static team starts by using one of the four basic tactics. For each encounter, the team will continue to employ the tactic used during the previous encounter if that encounter was won, but will switch to the next tactic if that encounter was lost. This strategy is closest to what human players do: they stick with a tactic as long as it works, and switch when it fails. This design makes the consecutive tactic the most difficult tactic to defeat.

5.2.5 Measuring Performance

In order to identify reliable changes in strength between teams, the notion of the 'turning point' is defined as follows. After each encounter the average fitness for each of the teams over the last ten encounters is calculated. The dynamic team is said to 'outperform' the static team at an encounter if the average fitness over the last ten encounters is higher for the dynamic team than for the static team. The turning point is the number of the first encounter after which the dynamic team outperforms the static team for at least ten consecutive encounters.

Figure 5.3 illustrates the turning point with a graph displaying the progression of the average team-fitness in a size-10 window (i.e., the values for the average team fitness for ten consecutive encounters) for both teams, in a typical test. The horizontal axis represents the encounters. Because of the size-10 window, the first values are displayed for encounter number 10. In this example at encounter number 29 the dynamic team outperforms the static team, and maintains its superior performance for ten encounters. Therefore, the turning point is 29. The absolute fitness values for the same typical test are displayed in Figure 5.4. Since after each encounter the fitness for one of the teams is zero, only the fitness for the winning team is displayed per encounter (the colour of the bar indicates which is the winning team).

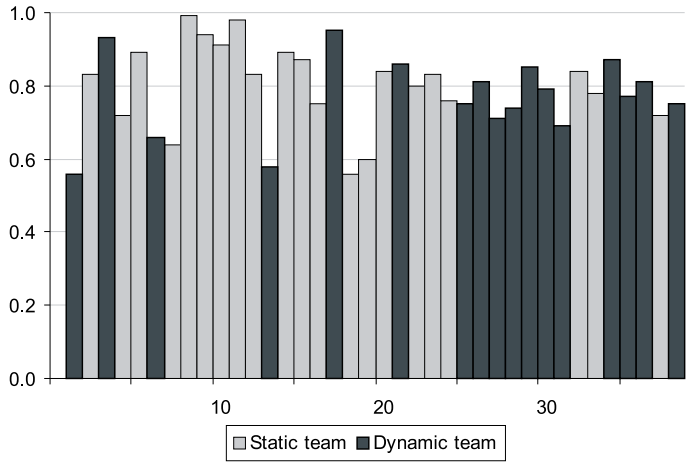

Figure 5.4: Absolute fitness $F(g)$ as a function of the encounter number.

Evidently, after encounter 25, the dynamic team wins more often than the static team. Note that, regardless how long training lasts, the dynamic team will never reach a point where it is able to win always, due to (i) the randomness inherent in the simulation, (ii) the variety of the scripts generated by dynamic scripting, and (iii) the effectiveness of the static tactics.

A low value for the turning point indicates good efficiency of dynamic scripting, since it indicates that the dynamic team consistently outperforms the static team within a few encounters only.

5.2.6 Efficiency-Validation Results

For each of the tactics I ran 100 tests to determine the average turning point. The results of these tests are presented in Table 5.1. The columns of the table represent, from left to right, (i) the name of the tactic, (ii) the average turning point, (iii) the standard deviation, (iv) the median, (v) the highest value for a turning point found, and (vi) the average of the five highest values.

The aim of the first experiment was to test the viability and efficiency of dynamic scripting. The achieved results show that dynamic scripting is both a viable, and a highly efficient technique (at least in the present domain of combat in CRPGs). For all tactics, dynamic scripting yields low turning points. In addition to this general observation, I give three more specific observations.

First, the 'disabling' tactic is easily defeated by the dynamic team. Apparently it is not a good tactic, because dealing with it requires little or no adaptation of the rulebase.

Second, the 'consecutive' tactic, which was argued to be closest to human-player

Tactic	Average	St.dev.	Median	Highest	Top 5
Offensive	58	35.0	53	314	155
Disabling	12	5.2	10	51	31
Cursing	137	333.6	35	1767	1461
Defensive	31	18.8	27	93	77
Random team	56	74.4	34	595	310
Random agent	53	67.0	27	398	289
Consecutive	72	100.3	47	716	424

Table 5.1: Turning-point values for dynamic scripting pitted against seven different tactics, averaged over 100 tests.

behaviour, is overall the most difficult to defeat with dynamic scripting.[2] Nevertheless, the dynamic-scripting technique is capable of defeating this tactic rather quickly, especially considering the fact that the rulebase started out with all weights being equal, while in an actual game the weights would be biased from the start to give the objectively better rules a higher selection probability.

Third, it is striking that for all tactics the average turning point is significantly higher than the median. The explanation is the rare occurrence of extremely high turning points. These so-called 'outliers' are explained by the high degree of randomness that is inherent to the simulated CRPG, and to games in general. A long run of encounters where pure chance drives the learning process away from an optimum (e.g., a run of encounters wherein the dynamic team is lucky and wins despite employing inferior tactics, or wherein the dynamic team is unlucky and loses despite employing good tactics) may place the rulebase in a state from which it has difficulty to recover. Due to the randomness inherent in games, such occasional long runs are unavoidable, but their probability of occurrence may be reduced. Two countermeasures against outliers are discussed in Section 5.3.

5.3 Outlier Reduction

The occasional occurrence of outliers withholds dynamic scripting from meeting the requirement of consistency. To reduce the number of outliers occurring with the application of dynamic scripting, I propose two countermeasures, namely (i) penalty balancing, and (ii) history fallback. The two countermeasures are explained in Subsections 5.3.1 and 5.3.2, respectively. The countermeasures are evaluated in an experiment, of which the results are presented in Subsection 5.3.3, and discussed in Subsection 5.3.4.

[2] At first glance the 'cursing' tactic might seem harder to defeat, but the median value shows that this is not the case; the 'cursing' tactic's high average is caused by its high susceptibility to outliers, which are discussed in Section 5.3

5.3.1 Penalty Balancing

The magnitude of the weight adaptation in a rulebase depends on a measure of the success (or failure) of the agent whose script is extracted from the rulebase. It is calculated according to equation 5.6. 'Penalty balancing' is balancing the magnitude of the maximum penalty P_{max} against the maximum reward R_{max}, to optimise speed and effectiveness of the adaptation process. The experimental results presented in Section 5.2 relied on a maximum penalty that was substantially smaller than the maximum reward (namely, $P_{max} = 30$ and $R_{max} = 100$). As stated in Subsection 5.2.3, the argument for the relatively small maximum penalty is that, as soon as a local optimum is found, the rulebase should be protected against degradation. However, when a sequence of undeserved rewards leads to wrong settings of the weights, recovering the appropriate weight values is hampered by a relatively low maximum penalty. Penalty balancing attempts to take this into account by balancing the need to recover from erroneous weight values against the risk of moving away from an optimum.

5.3.2 History Fallback

In the formulation of dynamic scripting in Section 5.1, the old weights of the rules in the rulebase are erased when the rulebase adapts. With history fallback all previous weights are retained in so-called 'historic rulebases'. When learning seems to be stuck in a sequence of rulebases that have inferior performance, it can 'fall back' to one of the historic rulebases that seemed to perform better.

Caution should be taken not to be too eager to fall back to earlier rulebases. The dynamic-scripting technique is quite robust, and learns from both successes and failures. Returning to an earlier rulebase means losing everything that was learned after that rulebase was generated. Furthermore, an earlier rulebase may have a high fitness due to chance, and returning to it might therefore have an adverse effect. It was empirically confirmend that the performance of dynamic scripting worsened when extended with a history-fallback mechanism that was eager to return to a previous rulebase. Therefore, history fallback should only be activated when there is a high probability that a truly inferior rulebase is replaced by a truly superior one.

The implementation of history fallback is as follows. The current rulebase R is used to generate scripts that control the behaviour of an agent during an encounter. After each encounter i, before the weight updates, all weight values from rulebase R are copied to historic rulebase R_i. With R_i are also stored: the team-fitness value $F(g)$, the agent-fitness value $F(a, g)$, and a number representing the so-called 'parent' of R_i. The parent of R_i is the historic rulebase whose weights were updated to generate R_i (usually the parent of R_i is R_{i-1}). A rulebase is considered 'inferior' when both its own fitness values and the fitness values of its N immediate ancestors, are low (i.e., below a threshold value T). A rulebase is considered 'superior' when both its own fitness values and the fitness values of its N immediate ancestors, are high (i.e., above T). If at encounter i we find that R_i is inferior, and in R_i's ancestry we find a historic rulebase R_j that is superior, the next parent used to generate the

current rulebase R will not be R_i but R_j. Because it is useless to return to a historic rulebase that has not yet learned, the mechanism only falls back to a rulebase R_j for $j > J$. In the experiments $N = 3$, $T = 0.4$, and $J = 10$ were used.

Though unlikely, with this mechanism it is still possible to fall back to a historic rulebase that does not perform well in the current situation, although it seemed to perform well in the past. While this will be discovered by the learning process soon enough, the risk of returning to such a rulebase over and over again should be minimised. I propose two different ways of avoiding this risk. The first is by simply not allowing the mechanism to fall back to a historic rulebase that is 'too old', but only allow it to fall back to the last M ancestors (in the experiment $M = 15$ was used). This is called 'limited-distance fallback' (LDF). The second is acknowledging that the agent-fitness value of a rulebase should not be too different from that of its direct ancestors. This is realised by propagating a newly calculated fitness value back through the ancestry of a rulebase, and factoring it into the fitness values for those ancestors. As a consequence, a rulebase that has children with low agent-fitness values will be assigned an agent-fitness value that is also small. This is called 'fitness-propagation fallback' (FPF). Both versions of history fallback allow dynamic scripting to recover earlier rulebases, that are truly better than the current one.

5.3.3 Outlier-Reduction Results

To test the effectiveness of penalty balancing and history fallback, I ran an experiment in the simulated CRPG. The experiment consisted of a series of tests, executed in a manner equal to the efficiency-validation experiment (5.2). I decided to use the 'consecutive' tactic for the static team, since this tactic is the most challenging for dynamic scripting. I compared nine different configurations, namely learning runs using maximum penalties $P_{max} = 30$, $P_{max} = 70$ and $P_{max} = 100$, combined with the use of no fallback (NoF), limited-distance fallback (LDF), and fitness-propagation fallback (FPF). All other parameters were set equal to the values used in the efficiency-validation experiment.

Table 5.2 gives an overview of the experimental results. The columns of the table represent, from left to right, (i) the value for P_{max}, (ii) the history-fallback mechanism used, (iii) the average turning point, (iv) the standard deviation, (v) the median, (vi) the highest value for the turning point, and (vii) the average of the five highest values.

Figure 5.5 shows histograms of the turning points for each of the series of tests. The turning points have been grouped in ranges of 25 different values. Each bar indicates the number of turning points falling within a range. Each graph starts with the leftmost bar representing the range $[0, 24]$. The rightmost bars in the topmost three graphs represent all turning points of 500 or greater (the other graphs do not have turning points in this range).

From Table 5.2 and Figure 5.5 I derive the following four observations. (i) Penalty balancing is a necessary requirement to reduce the number of outliers. All experiments that have a higher maximum penalty than the original $P_{max} = 30$ reduce the

P_{max}	Fallback	Average	St.dev.	Median	Highest	Top 5
30	NoF	72	100.3	47	716	424
30	LDF	99	229.3	49	2064	837
30	FPF	80	145.0	54	971	605
70	NoF	62	69.4	44	336	301
70	LDF	52	56.2	37	393	238
70	FPF	60	57.3	32	391	245
100	NoF	66	59.5	59	322	246
100	LDF	68	56.7	60	271	225
100	FPF	57	50.6	53	331	202

Table 5.2: Turning-point values for dynamic scripting pitted against the consecutive tactic, averaged over 100 tests.

number and magnitude of outliers.[3] (ii) There is no discernable difference in the effect of limited-distance fallback and the effect of fitness-propagation fallback. (iii) If penalty balancing is not applied, history fallback seems to have no effect or even an adverse effect. (iv) If penalty balancing is applied, history fallback has no adverse effect and may actually have a positive effect. One of the reasons why history fallback is so effective in combination with penalty balancing may be the following. In Subsection 5.3.1 it was stated that penalty balancing runs the risk of losing a discovered optimum due to chance. History fallback counteracts this risk.

As a final test, a combination of penalty balancing with $P_{max} = 70$ and limited-distance fallback was applied to all the different tactics available in the simulation environment. The results are summarised in Table 5.3. A comparison of Table 5.3 and Table 5.1 shows a significant, often very large reduction of the both the highest turning point and the average of the highest five turning points, for all tactics except for the 'disabling' tactic (note, however, that the increased turning points for the 'disabling' tactic are inconsequential, since the 'disabling' tactic already has the lowest turning points in both tables). Therefore, the results of the final test clearly support the positive effect of the two countermeasures against outliers.

5.3.4 Discussion of Outlier-Reduction Results

It is clear from the results in Table 5.2 that the number of outliers has been significantly reduced with the proposed countermeasures. However, exceptionally long learning runs still occur in the simulation experiments, even though they are rare, and less extreme than without the countermeasures. Does this mean that dynamic

[3] After the first publication of dynamic scripting by Spronck, Sprinkhuizen-Kuyper, and Postma (2003b), I was contacted by Dahlbom on the question how to apply dynamic scripting to real-time strategy games. Independently of the results reported by Spronck, Sprinkhuizen-Kuyper, and Postma (2004b), Dahlbom (2004) later arrived at a similar conclusion regarding the effect of penalty balancing on the reduction of outliers.

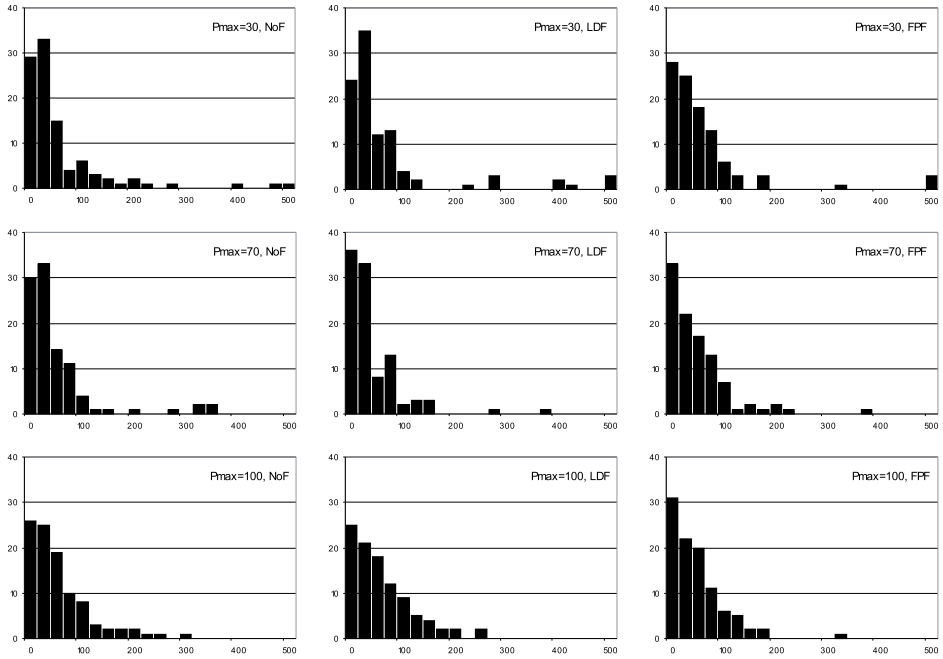

Figure 5.5: Histograms for the turning points in 100 tests, for the outlier-reduction experiment.

scripting, enhanced with the countermeasures, still does not meet the requirement of consistency?

I argue that the countermeasures do make dynamic scripting meet the requirement of consistency. The argument is twofold: (i) Because dynamic scripting is a non-deterministic technique, outliers can never be prevented completely. However, entertainment value of a game is guaranteed even if an outlier occurs, because dynamic scripting meets the requirement of effectiveness by design. (ii) Exceptionally long learning runs mainly occur because early in the process chance increases the wrong weights. This is not likely to happen in a rulebase with pre-initialised weights. When dynamic scripting is implemented in an actual game, the weights in the rulebase will not all start out with equal values, but they will be initialised to values that are already trained against commonly used tactics. This will not only prevent the occurrence of outliers, but also increase the speed of the dynamic scripting process, and provide history fallback with a likely candidate for a superior rulebase.

It should be noted that, besides as a target for the history-fallback mechanism, historic rulebases can also be used to store tactics that work well against a specific tactic employed by a human player. If human-player tactics can be identified, these rulebases can simply be reloaded when the player starts to use a particular tactic again after having employed a completely different tactic for a while.

Tactic	Average	St.dev.	Median	Highest	Top 5
Offensive	53	24.8	52	120	107
Disabling	13	8.4	10	79	39
Cursing	44	50.4	26	304	222
Defensive	24	15.3	17	79	67
Random team	51	64.5	29	480	271
Random agent	41	40.7	25	251	178
Consecutive	52	56.2	37	393	238

Table 5.3: Turning-point values for dynamic scripting pitted against different tactics, using $P_{max} = 70$ and limited-distance fallback, averaged over 100 tests.

5.4 Difficulty Scaling

For non-expert players, a game is most entertaining when it is challenging but beatable (Scott, 2002). To ensure that the game remains interesting, the issue is not for the computer to produce occasionally a weak move so that the human player can win, but rather to produce not-so-strong moves under the proviso that, on a balance of probabilities, they should go unnoticed (Iida, Handa, and Uiterwijk, 1995). 'Difficulty scaling' is the automatic adaptation of a game, to set the challenge that the game poses to a human player. When applied to game AI, difficulty scaling aims at achieving an 'even game', i.e., a game wherein the playing strength of the computer and the human player match.

Many games provide a 'difficulty setting', i.e., a discrete value that determines how difficult the game will be. The purpose of a difficulty setting is to allow both novice and experienced players to enjoy the appropriate challenge the game offers (Charles and Black, 2004). The difficulty setting commonly has some problematic issues, of which I indicate three. First, the setting is *coarse*, with the player having a choice between only a limited number of difficulty levels (usually three or four). Second, the setting is *player-selected*, with the player unable to assess which difficulty level is appropriate for his skills. Third, the setting has a *limited scope*, (in general) only affecting the computer-controlled agents' strength, and not their tactics. Consequently, even on a 'high' difficulty setting, the opponents exhibit similar behaviour as on a 'low' difficulty setting, despite their greater strength.

The three issues mentioned may be alleviated by applying dynamic scripting enhanced with an adequate difficulty-scaling mechanism. Dynamic scripting changes the computer's tactics to the way a game is played. As such, (i) it makes changes in small steps (i.e., it is not coarse), (ii) it makes changes automatically (i.e., it is not player-selected), and (iii) it affects the computer's tactics (i.e., it does not have a limited scope).

This section describes three different enhancements to the dynamic-scripting technique that let agents learn how to play an even game, namely (i) high-fitness penalising, (ii) weight clipping, and (iii) top culling. The three enhancements are

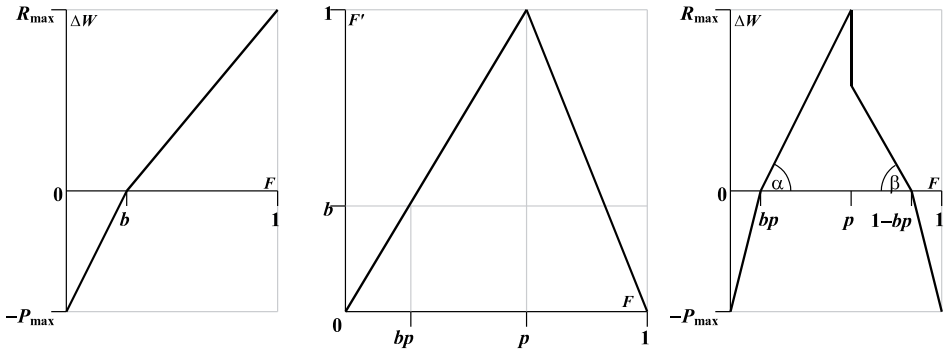

Figure 5.6: Comparison of the original weight-adjustment formula (left) and the high-fitness-penalising weight-adjustment formula (right), by plotting the weight adjustments as a function of the fitness value F. The middle graph displays the relation between F and F'.

explained in Subsections 5.4.1, 5.4.2, and 5.4.3, respectively. The enhancements are evaluated in an experiment, of which the results are presented in Subsection 5.4.4, and discussed in Subsection 5.4.5.

5.4.1 High-Fitness Penalising

The weight adjustment expressed in equation 5.6 gives rewards proportional to the fitness value: the higher the fitness, the higher the reward. To elicit mediocre instead of good behaviour, the weight adjustment can be changed to give highest rewards to mediocre fitness values, and lower rewards or even penalties to high fitness values. With high-fitness penalising the weight adjustment is expressed by formula 5.6, where F is replaced by F' defined as follows.

$$F' = \begin{cases} \dfrac{F}{p} & \{F \leq p\} \\ \dfrac{1-F}{p} & \{F > p\} \end{cases} \tag{5.7}$$

In this equation, F is the calculated fitness value, and $p \in [0.5, 1]$, $p > b$, is the reward-peak value, i.e., the fitness value that should get the highest reward. The higher the value of p, the more effective agent behaviour will be. Figure 5.6 illustrates the weight adjustment as a function of the original fitness (left), the mapping of F to F' (middle), and the weight adjustment as a function of the high-fitness-penalising fitness (right). Angles α and β are equal.

Since the optimal value for p depends on the tactic that the human player uses, it was decided to let the value of p adapt to the perceived difficulty level of a game,

as follows. Initially p starts at a value p_{init}. After every fight that is lost by the computer, p is increased by a small amount p_{inc}, up to a predefined maximum p_{max}. After every fight that is won by the computer, p is decreased by a small amount p_{dec}, down to a predefined minimum p_{min}. By running a series of tests with static values for p, I found that good values for p are found close to 0.7. Therefore, in the experiment I used $p_{init} = 0.7$, $p_{min} = 0.65$, $p_{max} = 0.75$, and $p_{inc} = p_{dec} = 0.01$.

5.4.2 Weight Clipping

During the weight updates, the maximum weight value W_{max} determines the maximum level of optimisation a learned tactic can achieve. A high value for W_{max} allows the weights to grow to large values, so that after a while the most effective rules will almost always be selected. This will result in scripts that are close to optimal. A low value for W_{max} restricts weights in their growth. This enforces a high diversity in generated scripts, most of which will be mediocre.

Weight clipping automatically changes the value of W_{max}, with the intent to enforce an even game. It aims at having a low value for W_{max} when the computer wins often, and a high value for W_{max} when the computer loses often. The implementation is as follows. After the computer wins a fight, W_{max} is decreased by W_{dec} per cent (but not lower than the initial weight value W_{init}). After the computer loses a fight, W_{max} is increased by W_{inc} per cent.

Figure 5.7 illustrates the weight-clipping process and the associated parameters. The shaded bars represent weight values for four arbitrary rules on the horizontal axis, numbered 1 to 4. After a fight, before weight adjustment, W_{max} is either increased by W_{inc} per cent, or decreased by W_{dec} per cent, depending on the outcome of the fight. After the change of W_{max}, in the figure the weight value for rule 4 is too low, so it is increased to W_{min} (the arrow marked 'a'). Similarly, the weight value for rule 2 is too high, so it is decreased to W_{max} (the arrow marked 'b'). As prescribed by dynamic scripting, after the weights are brought within the range $[W_{min}, W_{max}]$, the excess weights are redistributed again over all weights.

In the experiment I decided to use the same initial values as I used for the efficiency-validation experiment, i.e., I used $W_{init} = 100$, $W_{min} = 0$, and an initial value for W_{max} of 2000. W_{inc} and W_{dec} I both set to 10 per cent.

5.4.3 Top Culling

Top culling is quite similar to weight clipping. It employs the same adaptation mechanism for the value of W_{max}. The difference is that top culling allows weights to grow beyond the value of W_{max}. However, rules with a weight greater than W_{max} will not be selected for a generated script. Consequently, when the computer-controlled agents win often, the most effective rules will have weights that exceed W_{max}, and cannot be selected, and thus the agents will use weak tactics. Alternatively, when the computer-controlled agents lose often, rules with high weights will be selectable, and the agents will use strong tactics. So, while weight clipping achieves weak tactics

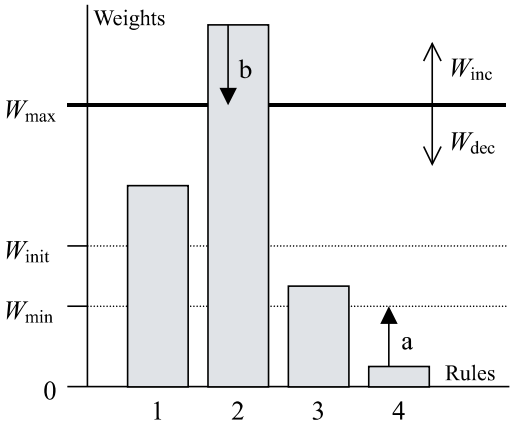

Figure 5.7: Weight-clipping and top-culling process and parameters.

by promoting diversity, top culling achieves weak tactics by removing access to the most effective domain knowledge.

In Figure 5.7, contrary to weight clipping, top culling will leave the value of rule 2 unchanged (the action represented by arrow 'b' will not be performed). However, rule 2 will be unavailable for selection, because its value exceeds W_{max}.

5.4.4 Difficulty-Scaling Results

To test the effectiveness of the three difficulty-scaling enhancements, I ran an experiment in the simulated CRPG. The experiment consisted of a series of tests, executed in the same way as the efficiency-validation experiment (Section 5.2). The experiment aimed at assessing the performance of a team controlled by the dynamic-scripting technique using a difficulty-scaling enhancement (with $P_{max} = 100$, fitness-propagation fallback, and all other parameters equal to the values used in the efficiency-validation experiment), against a team controlled by static scripts. If the difficulty-scaling enhancements work as intended, dynamic scripting will balance the game so that the number of wins of the dynamic team is roughly equal to the number of losses.

For the static team, I added an eighth tactic to the seven tactics described in Subsection 5.2.4, called the 'novice' tactic. The 'novice' tactic resembles the playing style of a novice CRPG player, who has learned the most obvious successful tactics, but has not yet mastered the subtleties of the game. While normally the 'novice' tactic will not be defeated by arbitrarily choosing rules from the rulebase, there are many different tactics that can be employed to defeat it, which the dynamic team will discover quickly. Against the 'novice' tactic, without a difficulty-scaling enhancement, the dynamic team's number of wins in general will greatly exceed its losses.

Tactic	Plain		High-Fitness Penalising		Weight Clipping		Top Culling	
	Avg.	Dev.	Avg.	Dev.	Avg.	Dev.	Avg.	Dev.
Offensive	61.2	16.4	**46.0**	15.1	**50.6**	9.4	**46.3**	7.5
Disabling	86.3	10.4	56.6	8.8	67.8	4.5	**52.2**	3.9
Cursing	56.2	11.7	42.8	9.9	**48.4**	6.9	**46.4**	5.6
Defensive	66.1	11.9	39.7	8.2	**52.7**	4.2	**49.2**	3.6
Novice	75.1	13.3	**54.2**	13.3	**53.0**	5.4	**49.8**	3.4
Random team	55.8	11.3	37.7	6.5	**50.0**	6.9	**47.4**	5.1
Random agent	58.8	9.7	44.0	8.6	**51.8**	5.9	**48.8**	4.1
Consecutive	**51.1**	11.8	34.4	8.8	**48.7**	7.7	**45.0**	7.3

Table 5.4: Experimental results of testing the difficulty-scaling enhancements to dynamic scripting on eight different tactics, averaged over 100 tests.

For each of the tactics, I ran 100 tests in which dynamic scripting was enhanced with each of the three difficulty-scaling enhancements, and, for comparison, also without difficulty-scaling enhancements (called 'plain'). Each test consisted of a sequence of 150 encounters between the dynamic team and the static team. Because in each of the tests the dynamic-scripting technique starts with a rulebase with all weights equal, the first 50 encounters were used for finding a balance of well-performing weights. I recorded the number of wins of the dynamic team over the last 100 encounters.

The results of these tests are displayed in Table 5.4. For each combination of tactic and difficulty-scaling enhancement the table shows the average number of wins over 100 tests, and the associated standard deviation. To be recognised as an even game, it was decided that the average number of wins over all tests must be close to 50. To take into account random fluctuations, in this context 'close to 50' means 'within the range [45,55]'.[4] In Table 5.4, all cell values indicating an even game are marked in bold font. From the table the following four results can be derived.

First, dynamic scripting without a difficulty-scaling enhancement ('plain') results in wins significantly exceeding losses for all tactics except for the 'consecutive' tactic (with a reliability $> 99.9\%$; Cohen, 1995). This supports the viability of dynamic scripting as a learning technique, and also supports the statement in Subsection 5.2.4 that the 'consecutive' tactic is the most difficult tactic to defeat. Note that the fact that, on average, dynamic scripting plays an even game against the 'consecutive' tactic is not because it is unable to defeat this tactic consistently, but because

[4] Deciding when a game can be called an 'even game' by observing the number of wins, seems to be comparable to deciding whether a coin is fair by observing a series of coin tosses, and thus be subject to a standard statistical evaluation to determine the range of the number of wins. However, the comparison is not apt. While coin tosses are random, the difficulty-scaling enhancements actively force a game to equal wins and losses. Imagine a coin that moves the centre-point of its weight after every toss.

dynamic scripting continues learning after it has reached a local optimum. Therefore, it can 'forget' what it previously learned, especially against an superior tactic like the 'consecutive' tactic.

Second, high-fitness penalising performs considerably worse than the other two enhancements. It cannot achieve an even game against six out of the eight tactics.

Third, weight clipping is successful in enforcing an even game in seven out of eight tactics. It does not succeed against the 'disabling' tactic. This is caused by the fact that the 'disabling' tactic is so easy to defeat, that even a rulebase with all weights equal will, on average, generate a script that defeats this tactic. Weight clipping can never generate a rulebase worse than 'all weights equal'.

Fourth, top culling is successful in enforcing an even game against all eight tactics.

Histograms for the tests with the 'novice' tactic are displayed in Figure 5.8. On the horizontal axis the number of wins for the dynamic team out of 100 fights is displayed. The bar length indicates the number of tests that resulted in the associated number of wins.

From the histograms the following result is derived. While, on average, all three difficulty-scaling enhancements manage to enforce an even game against the 'novice' tactic, the number of wins in each of the tests is much more 'spread out' for the high-fitness-penalising enhancement than for the other two enhancements. This indicates that the high-fitness penalising enhancement results in a higher variance of the distribution of won games than the other two enhancements. The top-culling enhancement seems to yield the lowest variance. This is confirmed by an approximate randomisation test (Cohen, 1995), which shows that against the 'novice' tactic, the variance achieved with top culling is significantly lower than with the other two enhancements (reliability $> 99.9\%$). I observed similar distributions of won games against the other tactics, except that against some of the stronger tactics, a few exceptional outliers occurred with a significantly lower number of won games. The rare outliers were caused by the fact that, occasionally, dynamic scripting requires more than 50 encounters to find a well-performing set of weights when playing against a strong static tactic.

In conclusion, the results show that, when dynamic scripting is enhanced with the top-culling difficulty-scaling mechanism, it meets the functional requirement of scalability.

5.4.5 Discussion of Difficulty-Scaling Results

Of the three different difficulty-scaling enhancements the top-culling enhancement is the best choice. It has the following three advantages: (i) it gives the most reliable results, (ii) it is easily implemented, and (iii) of the three enhancements, it is the only one that manages to force an even game against inferior tactics.

Obviously, the worst choice is the high-fitness-penalising enhancement. In an attempt to improve high-fitness penalising, some tests were performed with different ranges and adaptation values for the reward-peak value p, but these worsened the results. However, the possibility cannot be ruled out that with a different fitness

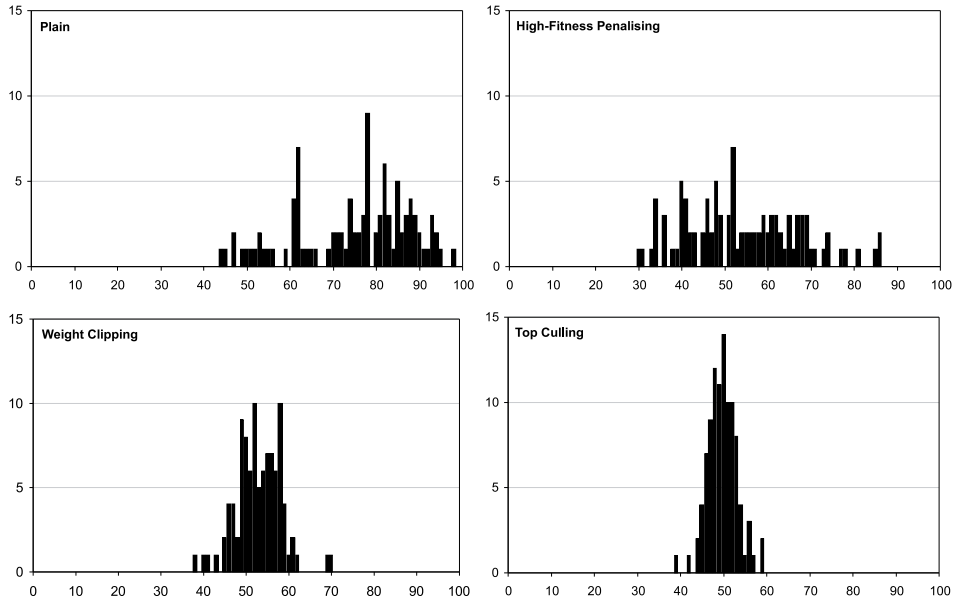

Figure 5.8: Histograms of 100 tests of the achieved number of wins in 100 fights, against the 'novice' tactic.

function high-fitness penalising will give better results.[5]

An additional possibility with the weight-clipping and top-culling enhancements is that they can also be used to set a different desired win-loss ratio, by changing the rates with which the value of W_{max} fluctuates. For instance, by using top culling with $W_{dec} = 30$ per cent instead of 10 per cent, leaving all other parameters unchanged, after 100 tests against the 'novice' tactic I derived an average number of wins of 35.0 with a standard deviation of 5.6. The histogram of this experiment is given in Figure 5.9.

Notwithstanding the successful results, a difficulty-scaling enhancement should be an optional feature in a game, that can be turned off by the player, for the following two reasons: (i) when confronted with an experienced player, the learning process should aim for superior tactics without interference from a difficulty-scaling enhancement, and (ii) some players will feel that attempts by the computer to force an even game diminishes their accomplishment of defeating the game, so they may prefer not to use it.

[5] In independent research (see footnote 3) Dahlbom (2004) applied dynamic scripting to a simulated real-time strategy game. He used a technique which he called 'fitness mapping' for difficulty scaling, for which he reported good results. Fitness mapping is similar to what I call 'high-fitness penalising' (Spronck, Sprinkhuizen-Kuyper, and Postma, 2004a), without dynamically changing the reward-peak value p.

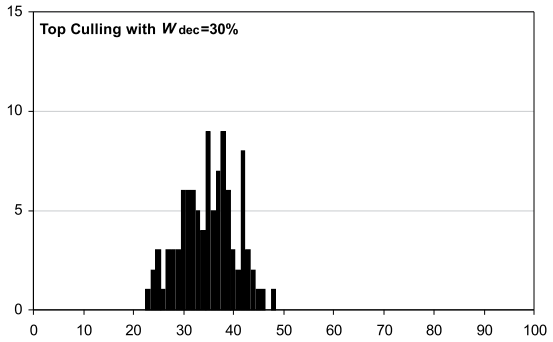

Figure 5.9: Histogram of the achieved number of wins over 100 tests against the 'novice' tactic, using dynamic scripting with the top-culling enhancement, with $W_{dec} = 30$ per cent.

5.5 Validation in Practice

To investigate whether the successful results achieved with dynamic scripting in a simulated CRPG hold in a practical setting, I decided to test the technique in an actual state-of-the-art commercial game. For this purpose, I chose the game NEVERWINTER NIGHTS (2002), developed by BioWare Corp. In this section I present the NEVERWINTER NIGHTS environment (5.5.1), the scripts and rulebases (5.5.2), the weight-update function (5.5.3), the tactics used by the static team (5.5.4), the results of an evaluation of dynamic scripting in NEVERWINTER NIGHTS (5.5.5), and a discussion of the results (5.5.6).

5.5.1 Neverwinter Nights

NEVERWINTER NIGHTS is a CRPG of a complexity similar to the BALDUR'S GATE games. A major reason for selecting NEVERWINTER NIGHTS for evaluating dynamic scripting is that the game is easy to modify and extend. It is delivered with a toolset that allows the user to develop completely new game modules. The toolset provides access to the scripting language and all the scripted game resources, including the game AI. While the scripting language is not as powerful as modern programming languages, I found it to be sufficiently powerful to implement dynamic scripting.

I implemented a small module in NEVERWINTER NIGHTS, similar to the simulated CRPG used previously. The module contains an encounter between a dynamic team and a static team of similar composition. As a result, the NEVERWINTER NIGHTS experiment is very similar to the CRPG simulation experiments described earlier. This is on purpose, because the present experiment is meant to demonstrate that the simulation results can be repeated in a commercially available game. In contrast, Chapter 6 will demonstrate the general applicability of dynamic scripting.

The testing environment is illustrated in Figure 5.10. Each team consists of a

Figure 5.10: A fight between two teams in NEVERWINTER NIGHTS.

fighter, a rogue, a priest, and a wizard of equal experience level. In contrast to the agents in the simulated CRPG, the inventory and spell selections in the NEVERWINTER NIGHTS module cannot be changed, due to the toolset lacking functions to achieve such modifications. This has a restrictive impact on the tactics. Details of the module are found in Appendix B, Section B.1.

5.5.2 Scripts and Rulebases

To facilitate the development of new game modules, the default game AI in NEVERWINTER NIGHTS is implemented in a very general way, suitable for agents of all classes and levels (e.g., it does not refer to casting of a specific magic spell, but to casting of spells from a specific class). It distinguishes between about a dozen agent classes. For each agent class it sequentially checks a number of environmental variables and attempts to generate an appropriate response. The behaviour generated is not completely predictable, because it is partly probabilistic. Details of the NEVERWINTER NIGHTS game AI are found in Appendix B, Section B.2.

For the implementation of the dynamic-scripting technique, first the rules employed by the default game AI were extracted, and then entered in every appropriate rulebase. To these standard rules several new rules were added. The new rules were similar to the standard rules, but slightly more specific, e.g., referring to specific enemies instead of referring to a random enemy. Additionally, a few 'empty' rules were added, which, if selected, allow the game AI to decrease the number of effective rules. Priorities were set similar to the priorities used in the simulated CRPG.

Note that since the rules extracted from the default game AI are generalised, the rules used by dynamic scripting are generalised too. The use of generalised rules in the rulebase has the advantage that the rulebase gets trained for generating AI for agents of any experience level.

The size of the scripts for both a fighter and a rogue was set to five rules (the same as the number of rules of the fighter in the simulated CRPG), which were selected out of rulebases containing 21 rules. The size of the scripts for both a priest and a wizard was set to ten rules (the same as the number of rules of the wizard in the simulated CRPG), containing 55 rules and 49 rules, respectively. To the end of each script a call to the default game AI was added, in case no rule could be activated. Details of the rulebases are found in Appendix B, Section B.3.

5.5.3 Weight-Update Function

The weight adjustment mechanism used in NEVERWINTER NIGHTS was similar to the mechanism used in the simulated CRPG (5.2.3). I decided to differ slightly from the implementation of these functions in the simulation, mainly to avoid problems with the NEVERWINTER NIGHTS scripting language, and to allow varying team sizes. These changes are not critical for the performance of dynamic scripting, since the fitness functions only need to provide a general indication of the measure of success of a team and its agents.

The team-fitness $F(g)$, which yields a value in the range [0,1], was defined as follows.

$$F(g) = \begin{cases} 0 & \{g \text{ lost}\} \\ \frac{1}{5} + \sum_{c \in g, h_T(c) > 0} \frac{2}{5N_g} \left(1 + \frac{h_T(c)}{h_0(c)} \right) & \{g \text{ won}\} \end{cases} \tag{5.8}$$

All variables in this equation were defined as those in equation 5.1. The agent-fitness $F(a, g)$, which yields a value in the range [0,1], was defined as follows.

$$F(a,g) = \frac{1}{2} F(g) + \frac{1}{2} \begin{cases} \min\left(\frac{2D(a)}{D_{max}}, \frac{3}{5} \right) & \{h_T(a) \leq 0\} \\ \frac{3}{5} + \frac{2\, h_T(a)}{5\, h_0(a)} & \{h_T(a) > 0\} \end{cases} \tag{5.9}$$

All variables in this equation were defined as those in equations 5.2 to 5.5.

Weight adjustment was implemented according to equation 5.6, with all parameter values as in the efficiency-validation experiment, except for the maximum penalty P_{max}, which was set to 50. Furthermore, rules in the script that were not executed during the encounter, instead of being treated as not being in the script at all, were assigned half the reward or penalty received by the rules that were executed. The main reason for this is that if there were no rewards and penalties for the non-executed rules, the empty rules would never get rewards or penalties.

5.5.4 Tactics

In our experiment three different tactics were used for the static team, all based on the default game AI, implemented by the NEVERWINTER NIGHTS developers. The three tactics are the following.

AI 1.29: AI 1.29 is the default game AI used in NEVERWINTER NIGHTS version 1.29. This version of NEVERWINTER NIGHTS was used for the earliest tests.

AI 1.61: AI 1.61 is the default game AI used in NEVERWINTER NIGHTS version 1.61. This version of NEVERWINTER NIGHTS was used for the later tests. Between version 1.29 and 1.61 the game AI was significantly improved by the game developers.

Cursed AI: A 'cursed' version of AI 1.61 was created. With cursed AI in 20 per cent of the encounters the game AI deliberately misleads dynamic scripting into awarding high fitness to purely random tactics, and low fitness to tactics that have shown good performance during earlier encounters.

5.5.5 Neverwinter Nights Results

Table 5.5 summarises the results from the repetition of (parts of) the efficiency-validation experiment and the outlier-reduction experiment in the NEVERWINTER NIGHTS environment. The columns of the table represent, from left to right, (i) the tactic used, (ii) the fallback mechanism used, (iii) the number of tests executed,[6] (iv) the average turning point, (v) the standard deviation, (vi) the median, (vii) the highest value for the turning point, and (viii) the average of the five highest values. No tests were performed with penalty balancing, since already in the earliest experiments with NEVERWINTER NIGHTS higher maximum penalties were used than in the simulated CRPG. From the results in Table 5.5 the following two conclusions are derived.

First, since the achieved turning points in all tests are (very) low, dynamic scripting meets the requirement of efficiency easily.

Second, history fallback has little or no effect on the results. However, since even 'cursed AI' does not cause significantly increased turning points, it seems that dynamic scripting in NEVERWINTER NIGHTS is so robust that remote outliers do not occur. Therefore, countermeasures against outliers are not needed, and dynamic scripting in NEVERWINTER NIGHTS meets the requirement of consistency without special measures.

The results achieved with the top-culling enhancement were also validated in NEVERWINTER NIGHTS. Without top culling, in ten tests dynamic scripting achieved

[6] The number of tests for the NEVERWINTER NIGHTS experiment is lower than for the simulation experiment, where I performed 100 tests for each configuration. Since the NEVERWINTER NIGHTS developers stated that it was not possible to increase the speed of the game execution, a test lasted 8 hours on average (for the fitness-scaling tests even 24 hours on average). To limit the time needed to do the tests, I decided to be satisfied with a number of tests sufficient to obtain statistically sound results.

Tactic	Fallback	Tests	Avg.	St.dev.	Median	Highest	Top 5
AI 1.29	NoF	50	21	8.8	16	101	58
AI 1.61	NoF	31	35	18.8	32	75	65
AI 1.61	FPF	30	32	21.8	24	104	71
Cursed AI	NoF	21	33	21.8	24	92	64
Cursed AI	FPF	21	32	28.1	18	115	69

Table 5.5: Turning-point values for dynamic scripting in NEVERWINTER NIGHTS.

an average number of 79.4 wins out of 100 fights, with a standard deviation of 12.7. With top culling, in ten tests dynamic scripting achieved an average number of 49.8 wins out of 100 fights, with a standard deviation of 3.4. The results clearly support that dynamic scripting, enhanced with top culling, meets the requirement of scalability.

5.5.6 Discussion

The NEVERWINTER NIGHTS experiment supports the results achieved with dynamic scripting in a simulated CRPG. Comparison of all results even seems to indicate that dynamic scripting performs better in NEVERWINTER NIGHTS than in the simulated CRPG. This is caused by the fact that the default game AI in NEVERWINTER NIGHTS is designed to be effective for all agents that can be designed in the toolset. Since it is not specialised, for most agents it is not optimal. Therefore, there is a great variety of tactics that can be used to deal with it, which makes it fairly easy for dynamic scripting to discover a successful counter-tactic.

In general, the more effective the tactic against which dynamic scripting is tested, the longer it will take for dynamic scripting to gain the upper hand. Moreover, because dynamic scripting is designed to generate a wide variety of tactics (in compliance with the requirement of variety), it will never gain the upper hand if the tactic against which it is pitted is so strong that there are *very* few viable counter-tactics. Against human players, this means that dynamic scripting will achieve the most satisfying results against non-expert players.

In a game that allows the design of 'super-tactics', which are almost impossible to defeat, dynamic scripting may not give satisfying results when used against expert players who know and use these super-tactics. However, *every* machine-learning technique will require more computational resources finding rare solutions than finding ubiquitous solutions. Therefore, against super-tactics, instead of using an online machine-learning technique, in general it will be more effective to use counter-tactics that have been trained against these super-tactics in an offline-learning process. It should be noted that the existence of super-tactics in a game is actually an indication of bad game-design, because they make the game too hard when employed by the computer, and they make the game too easy when employed by the human player.

5.6 Chapter Summary

By design, dynamic scripting meets the requirements of speed, effectiveness, robustness, clarity, and variety. In Section 5.2 it was shown that it meets the requirement of efficiency. In Section 5.3 it was shown that by applying penalty balancing, possibly combined with history fallback, dynamic scripting meets the requirement of consistency. In Section 5.4 it was shown that by applying top culling, dynamic scripting meets the requirement of scalability. The results achieved in a simulated CRPG were confirmed in the state-of-the-art CRPG NEVERWINTER NIGHTS. Therefore it may be concluded that dynamic scripting meets all eight requirements specified in Subsection 2.3.4, and thus can be applied in actual commercial games for the implementation of online adaptive game AI.

Chapter 6

Professional Adaptive Game AI

In the scale of destinies, brawn will never weigh as much as brain.
— James Russell Lowell (1819–1891).

This chapter[1] discusses how adaptive game AI is to be applied by professional game developers. Section 6.1 describes the game-development process, and indicates at which stages of the process adaptive game AI must be taken into account. While the offline application of adaptive game AI is relatively risk-free, game developers will only consider applying it online if it is of high reliability. A procedure is proposed to increase the reliability of online adaptive game AI by using offline adaptive game AI. The procedure is illustrated in Sections 6.2 to 6.4. Section 6.2 discusses adaptive game AI in a Real-Time Strategy (RTS) game. In Section 6.3 improved tactics for the game are generated with offline evolutionary game AI. In Section 6.4 the derived results are used to improve the reliability of the adaptive game AI introduced in Section 6.2. Section 6.5 discusses to what extent the investigated techniques can be accepted by game developers. A summary of the chapter is provided in Section 6.6.

6.1 Game Development and Adaptive Game AI

This section describes how adaptive game AI can be integrated in the game-development process. It discusses the game-development process (6.1.1), the stages of the process that are affected by adaptive game AI (6.1.2), and how offline adaptive game AI can be used to increase the reliability of online adaptive game AI (6.1.3).

6.1.1 The Game-Development Process

Crawford (1984) describes the game-development process as consisting of the following seven phases.[2]

[1] Sections 6.2 to 6.4 of this chapter are based on a paper by Ponsen and Spronck (2004).

[2] I replaced some of the terms used by Crawford (1984) with terms that are more common nowadays.

Concept: The 'concept' phase consists of setting a topic and a goal for a game. Each game must have a goal, that is expressed in terms of the effect the game has on human players. Setting a clear goal at the start of the game-development process supports game designers in taking decisions, especially when trade-offs between features must be considered.

Pre-production: After choosing a goal and a topic for a game, research must be done into the game's background, to give designers a feeling for the game's scope. This is an exploratory phase, in which little is put on paper.

Design: In the 'design' phase, designers create documents outlining three interdependent structures: (i) the I/O structure, (ii) the game structure, and (iii) the program structure. The I/O structure describes the game's interface, with respect to both input and output. The game structure describes how the game's goal and topic translate into game elements, to be experienced and manipulated by human players. The program structure describes how the I/O structure and game structure are translated into a real product.

Pre-development: In the 'pre-development' phase, the design documents are translated into a detailed technical design of the game.

Development: In the 'development' phase the game is implemented (which includes game debugging). Crawford (1984) calls this "the easiest of all phases". His argument is that "[p]rogramming itself is straightforward and tedious work, requiring attention to detail more than anything else." At the time he wrote this, it was certainly true, since games were much simpler then than they are today. Whether his statement is true for a modern game depends on how innovative and competitive the game intends to be.

Quality Assurance: 'Quality assurance', also referred to as 'playtesting', is meant to polish and refine the game design. Often during this phase fundamental flaws are discovered, that require major changes to the design or implementation.

Post-mortem: After the game has been deployed, the 'post-mortem' phase starts. Reactions of reviewers and the gaming public are measured. Nowadays, for most games during the 'post-mortem' phase one or more 'patches' are released, to resolve design and programming mistakes discovered only after a game's publication.

6.1.2 Integrating Adaptive Game AI

Before the late 1990s, game AI only became an issue late in the 'development' phase. However, since game AI has become an element of competition between game developers, as early as in the 'design' phase attention is given to game AI (Champandard, 2004). When adaptive game AI is introduced in a game, it affects the game-development process in even earlier phases, as explained below.

Since adaptive game AI is still new for published games, its introduction in a game will not be taken lightly. In particular online adaptive game AI has a major impact on the game-play experience of the human players. Since online adaptive game AI will be a unique selling point of a game, it becomes one of the game's goals. Therefore, the decision to include online adaptive game AI is taken in the 'concept' phase. This will remain the case until adaptive game AI becomes a proven technique that most games developers include by default.

For both offline and online adaptive game AI, the 'design' phase will be used to determine exactly what can be learned, and how the learning process is integrated into the game engine. In the 'pre-development' phase, detailed data structures are designed that store parameters used by the adaptive game AI. During the 'development' phase, the adaptive game AI is implemented.

With offline adaptive game AI, during the 'quality assurance' phase the game AI can be fine-tuned, in two ways. The first way is to let the manually-designed game AI play the game against offline adaptive game AI, to detect shortcomings and alternative tactics, as was discussed in Section 4.1. The second way is to store the tactics that are used by the playtesters, after which offline adaptive game AI is used to play against the stored tactics that playtesters seem to use often, to detect ways of defeating them.

For online adaptive game AI, special care must be taken during the 'quality assurance' phase to test the effect the adaptive mechanism has on the behaviour of the computer-controlled agents. Since the agents adapt to the human player, the human player has plenty opportunities to 'mess' with the game AI while playing the game. During the 'quality assurance' phase, it must be ascertained that the adaptive game AI meets the four computational and four functional requirements specified in Subsection 2.3.4. Adaptive game AI that meets all eight requirements is called 'reliable'. Game publishers can rest assured that the quality of reliable adaptive game AI is guaranteed, even against human players that deliberately try to exploit the adaptation process to elicit inferior game AI. However, because adaptive game AI is not static, the game developers must take into account that the 'quality assurance' phase for a game will take longer with than without adaptive game AI.

6.1.3 Combining Offline and Online Adaptive Game AI

To ensure the reliability of online adaptive game AI, it must incorporate a sufficient amount of correct prior domain knowledge (Manslow, 2002). However, if the incorporated domain knowledge is incorrect or insufficient, online adaptive game AI will not be reliable, and unable to generate satisfying results. If a combination of offline and online game AI is available during the 'quality assurance' phase, offline adaptive game AI can be used to increase the reliability of online adaptive game AI by improving the domain knowledge. To this end, I propose a procedure consisting of the following three steps.

1. *Online adaptation*: During the 'quality assurance' phase, online adaptive game AI is used against the playtesters and against manually-designed game AI, as

was shown in Chapter 5. The adaptive game AI will improve itself to generate successful tactics, that are hard to defeat.

2. *Offline adaptation*: Offline adaptive game AI is used to discover new tactics that can deal with the best results found by online adaptive game AI, and with the manually-designed tactics that online adaptive game AI was unable to deal with, as was shown in Section 4.1.

3. *Improving*: The tactics discovered with offline adaptive game AI are analysed, and the results of the analysis are used to improve the domain knowledge employed by online adaptive game AI. The improved online adaptive game AI should be better able to deal with strong human player tactics, and should be more efficient in finding tactics of a desired effectiveness. Step 1 can be repeated to validate the improvements. If necessary all three steps can be repeated to further improve the domain knowledge.

In the following three sections, the effectiveness of the procedure is demonstrated.[3]

6.2 Dynamic Scripting in an RTS Game

The first step in combining online and offline adaptive game AI is the implementation and use of online adaptive game AI. The most complex game AI is encountered in CRPGs and in strategy games (2.2.2). Chapter 5 already showed that dynamic scripting can be successfully applied to a CRPG. To demonstrate the general applicability of dynamic scripting, for the experiment described in the present chapter it was decided to apply dynamic scripting to a Real-Time Strategy (RTS) game. In the experiment, dynamic scripting is evaluated against several static tactics, to determine to what extent it is able to defeat the static tactics.

Subsection 6.2.1 introduces RTS games and the WARGUS environment used for the experiment. Subsection 6.2.2 describes the implementation of dynamic scripting in WARGUS. Subsection 6.2.3 discusses the evaluation of dynamic scripting in WARGUS. Subsection 6.2.4 presents the achieved results.

6.2.1 RTS Games

RTS games are simple military simulations (often called 'war games') that allow the human player to control a 'civilisation' on a map. Typically, a civilisation consists of buildings, technology, and armies. Armies consist of 'units' of several different types. A unit is an object that separately moves on a game's map, under the control of either a human player or the computer. A unit is different from an agent, in that a unit does not take autonomous decisions. All decisions are taken by the human player, or the centralised game AI used by the computer.

[3] This demonstration is based on the work by Ponsen (2004), which was performed in collaboration with and under supervision of the author.

The goal that an RTS game assigns to a human player is to defeat all opposing civilisations. Usually, defeating a civilisation equates eliminating all armies of the civilisation. In most RTS games, the key to winning lies in efficiently collecting and managing resources, and appropriately distributing the resources over the various game elements. Typical game elements in RTS games include the construction of buildings, the research of new technologies, and combat.

Game AI is of critical importance to RTS games. It determines the tactics of the civilisations controlled by the computer, including the management of resources. Designing game AI for RTS games is particularly challenging for game developers, because of two reasons: (i) RTS games are complex, i.e., a wide variety of tactics can be employed, and (ii) decisions have to be made under severe time constraints. Buro (2003b) calls RTS games "an ideal test-bed for real-time AI research".

Game AI in RTS games is global, i.e., it determines all decisions for a civilisation over the course of the whole game.[4] For RTS games, Ramsey (2004) describes a multi-tiered game-AI framework, which consists of different managers for different tasks. Five examples of managers are (i) a 'build manager' that is responsible for placement of structures and towns, (ii) a 'resource manager' that is responsible for gathering resources, (iii) a 'research manager' that selects new technologies based on their usefulness and cost, (iv) a 'combat manager' that is responsible for conscripting and deploying military units, and (v) a 'civilisation manager', that coordinates the interaction between the other managers. In practice, the managers are often combined in one game-AI script, which defines a strategy.

Because of the high complexity of the game AI of RTS games, usually the game AI employs a goal-directed approach (Harmon, 2002). The final goal for the game AI is to win the game, but this goal is too complicated to address directly. Therefore, the game AI aims at achieving subgoals, that can be considered successful steps on the road to achieving the final goal. Examples of subgoals are 'expanding the terrain under control' and 'disabling the opponent's resource gathering'. Usually, the game AI is enhanced with a variety of domain-specific tactics, which may increase the entertainment experienced by human players (Kent, 2004).

Contrary to publishers of CRPGs, publishers of RTS games have not yet released game engines that allow replacement of the game AI by an adaptive mechanism (Buro, 2004). Therefore, in the present context, an open-source game was selected to experiment with online and offline adaptive game AI in RTS games.

The game selected is WARGUS, illustrated in Figure 6.1. WARGUS is a faithful open-source clone of the game WARCRAFT II, developed by Blizzard. WARCRAFT II was first released in 1995, and re-released in 1999. While its graphics are not to up to today's standards, its game-play can still be considered state of the art. While WARCRAFT II and WARGUS allow conflicts between more than two civilisations, for the experiments described here, the number of civilisations on a map was limited to two. A game-AI script for WARGUS determines a complete strategy for a whole game. Details of the WARGUS game AI are found in Appendix C.

[4] Depending on the level of detail of an RTS game, it may also include local game AI, which controls unit behaviour. However, in strategy games the local game AI is trivial compared to the global game AI.

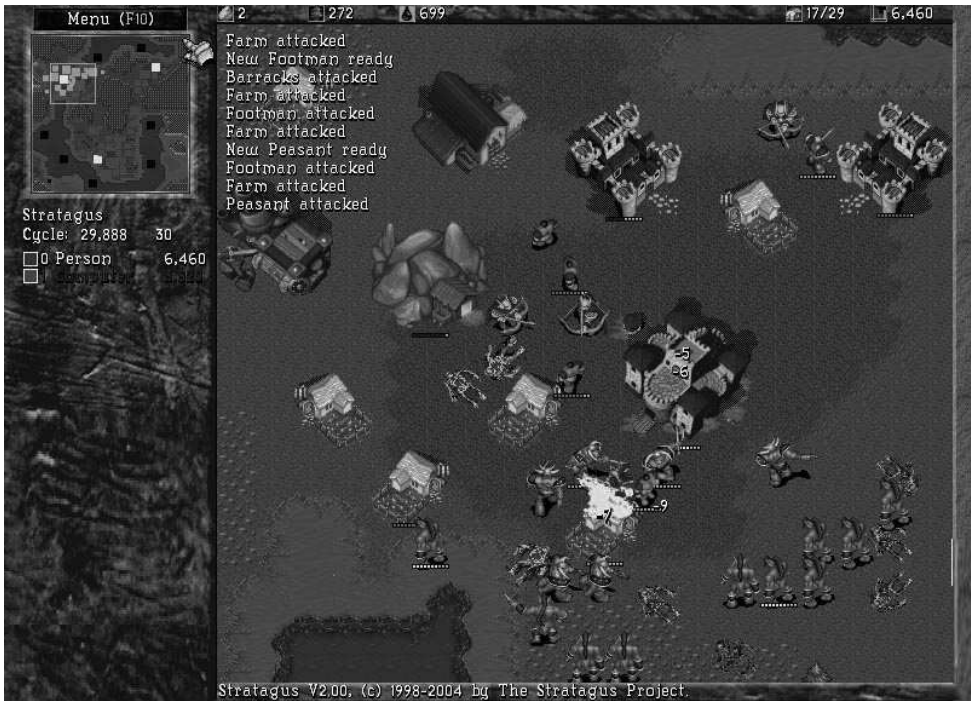

Figure 6.1: WARGUS.

6.2.2 Dynamic Scripting in Wargus

The design of dynamic scripting for RTS games has a major difference with dynamic scripting for CRPGs, as discussed in Chapter 5. While dynamic scripting for CRPGs employs different rulebases for different agent classes in the game, the RTS implementation of dynamic scripting employs different rulebases for different 'states' of the game. A 'state' of an RTS game is a game situation that the game-AI designer typifies as fundamentally different from other game situations. The reason for the deviation from the CRPG implementation of dynamic scripting is that the tactics that a civilisation can use in an RTS game depend on the current military, technological, and economical situation of the civilisation. Thus, rules that deserve high weights in one state, may not deserve high weights in another state. For instance, attacking with weak units might be the only viable choice in early game states, while in later game states, when strong units are available, usually weak units will have become useless.

In WARGUS the availability of different unit types and research options determines mainly what tactics are possible. The available buildings determine the unit types that can be trained, and the possibilities for research. Therefore, an obvious choice for defining different game states is by the buildings that have been con-

structed. Consequently, the construction of a building that allows the training of unit types that were previously unavailable, or that allows new research, spawns a state transition.

The twenty states for WARGUS, and the possible state transitions, are illustrated in Figure 6.2. In the figure, each box represents a state. Inside a box the buildings that are available are listed. The arrows between boxes, labelled with a building that is constructed, represent state transitions. Note that a civilisation starts out with a 'town hall' and with 'barracks' already available. Note also that buildings that do not allow the training of new unit types, new research, or the construction of new buildings, are left out of the figure.

For WARGUS, dynamic scripting was implemented as follows. To generate a new game-AI script, dynamic scripting starts by randomly selecting rules for the first state, from the rulebase corresponding to the first state. When a rule is selected that spawns a state transition, from that point on rules will be selected for the new state, using the rulebase corresponding to the new state. To avoid monotone behaviour, each rule is restricted to be selected only once per state. Rule selection continues, until either a total of N rules has been selected, or until a final state is reached from which no state transition is possible. For the final state (which, as can be observed in Figure 6.2, is state number 20), a maximum of N_{end} rules is selected. At the end of a script, a manually-designed group of commands is attached that initiate continuous attacks against the opposing civilisation.

In the experiment the values $N = 100$ and $N_{end} = 20$ were used. The value for N is similar to the size of the scripts created by the WARGUS developers. The value for N_{end} is largely irrelevant, since only in rare cases a game lasts until the final state.

To design rules for the rulebases, domain knowledge was acquired from strategy guides for WARCRAFT II. Fifty rules were defined this way, divided into four basic categories, namely (i) build rules (12 rules, for constructing buildings), (ii) research rules (9 rules, for acquiring new technologies), (iii) economy rules (4 rules, for gathering resources), and (iv) combat rules (25 rules, for military activities). To create rulebases for the twenty states, each rule was copied to all rulebases for states in which the rule can be executed.[5] This resulted in each of the rulebases containing between 21 and 42 rules. Details of the rulebases are supplied in Appendix C, Subsection C.5.1.

Because there are separate rulebases for each state, the size of weight updates is determined mainly by a so-called 'state fitness', i.e., an evaluation of performance of the game AI for each separate state. To recognise the importance of winning or losing the game, weight updates also take into account a so-called 'overall fitness', i.e., an evaluation of the performance of the game AI for the game as a whole. The use of both fitness functions for the weight updates increases the efficiency of the learning mechanism (Manslow, 2004).

A civilisation that uses dynamic scripting is called a 'dynamic civilisation'. The

[5] For instance, since in WARGUS a 'castle' is a prerequisite for building an 'airport', and since a civilisation only needs one 'airport', the rule 'build airport' is only included in rulebases for states in which a 'castle' is available, and in which an 'airport' has not been built yet.

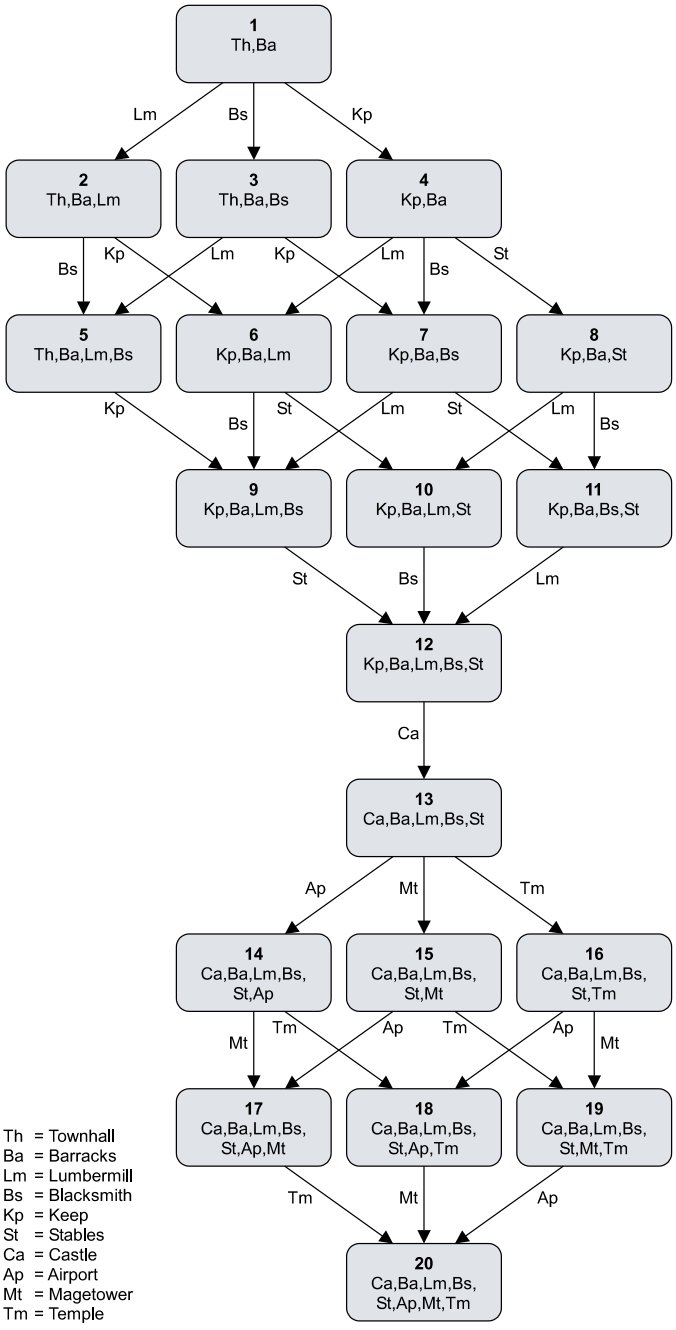

Figure 6.2: Game states in WARGUS.

state-fitness function F_i for state i, $i \in \mathbb{N}/\{0\}$, for dynamic civilisation d is defined as follows.

$$F_i = \frac{S_{d,i}}{S_{d,i} + S_{c,i}} - \frac{S_{d,i-1}}{S_{d,i-1} + S_{c,i-1}} \tag{6.1}$$

In this equation, $S_{d,x}$ represents the score of the dynamic civilisation after state x, $S_{c,x}$ represents the score of the civilisation opposing d after state x, $S_{d,0} = 0$, and $S_{c,0} = 1$. The score is a value that measures the success of a civilisation up to the moment the score is calculated.

The overall-fitness function F_∞ for dynamic civilisation d yields a value in the range $[0, 1]$. It is defined as follows.

$$F_\infty = \begin{cases} \min\left(\dfrac{S_{d,L}}{S_{d,L} + S_{c,L}}, b\right) & \{d \text{ lost}\} \\ \max\left(\dfrac{S_{d,L}}{S_{d,L} + S_{c,L}}, b\right) & \{d \text{ won}\} \end{cases} \tag{6.2}$$

In this equation, $S_{d,x}$ and $S_{c,x}$ are as in equation 6.1, L is the number of the state in which the game ended, and $b \in \langle 0, 1\rangle$ is the break-even value. At the break-even point, weights remain unchanged.

The score function is domain dependent, and should successfully reflect the relative strength of the two opposing civilisations in the game. For WARGUS, the score $S_{x,y}$ for civilisation x after state y is defined as follows.

$$S_x = C_m\, M_{x,y} + (1 - C_m)\, B_{x,y} \tag{6.3}$$

In this equation, for player x after state y, $M_{x,y}$ represents the 'military points' scored, i.e., the number of points awarded for killing units and destroying buildings, and $B_{x,y}$ represents the 'building points' scored, i.e., the number of points awarded for conscripting units and constructing buildings. $C_m \in [0, 1]$ represents the weight given to the military points in the fitness. Since experience indicates that military points are a better indication for the success of a tactic than building points, C_m was set to 0.7.

After each game, the weights of all rules employed are updated. Weight values are bounded by a range $[W_{min}, W_{max}]$. A new weight value is calculated as $W + \triangle W$, where W is the original weight value, and the weight adjustment $\triangle W$ is expressed by the following formula.

$$\triangle W = \begin{cases} -P_{max}\left(C_{end}\,\dfrac{b - F_\infty}{b} + (1 - C_{end})\,\dfrac{b - F_i}{b}\right) & \{F_\infty < b\} \\ R_{max}\left(C_{end}\,\dfrac{F_\infty - b}{1 - b} + (1 - C_{end})\,\dfrac{F_i - b}{1 - b}\right) & \{F_\infty \geq b\} \end{cases} \tag{6.4}$$

In this equation, $R_{max} \in \mathbb{N}$ and $P_{max} \in \mathbb{N}$ are the maximum reward and maximum penalty respectively, F_∞ is the overall fitness, F_i is the state fitness, for the state

corresponding to the rulebase containing the weight, and b is the break-even point. $C_{end} \in [0, 1]$ represents the fraction of the weight adjustment that is determined by the overall fitness. Since it can be expected that rulebases for different states will become successful at different times, the contribution of the state fitness F_i to the weight adjustment should be larger than the contribution of the overall fitness F_∞. Moreover, it is desirable that, even if a game is lost, rulebases for states where performance was successful are not punished (too much). Therefore, C_{end} was set to 0.3.

To keep the sum of all weight values in a rulebase constant, weight changes are executed through a redistribution of all weights in the rulebase. In the experiment, the values $W_{min} = 25$, $W_{max} = 1250$, $R_{max} = 200$, $P_{max} = 175$, and $b = 0.5$ were used. These values were determined to give good results during preliminary tests. The value of 0.5 for b is the only logical choice, since at this value the scores for the two civilisations are equal, indicating equal performance for both of them.

Note that it can be argued that, since the dynamic-scripting implementation in WARGUS executes weight updates only after a game has been played, the described adaptive game AI is actually an offline mechanism. However, an RTS game typically consists of a series of so-called 'levels', where each level is equivalent to a game as discussed above, i.e, civilisations start with little, and have to expand their territories and defeat all opposing armies, before moving on to the next level. Therefore, the described adaptive game AI learns during the playing of a full RTS game. Furthermore, with a fitness function that only uses state fitness, and with game AI generated for each state on the fly, learning can even take place during the playing of a level, if states can be revisited, or if the human player is pitted against multiple computer-controlled civilisations.

6.2.3 Evaluating of Dynamic Scripting in Wargus

Similar to the experiments reported in Chapter 5, the performance of dynamic scripting in WARGUS was evaluated by testing a dynamic civilisation against a civilisation using manually-designed game AI, called a 'static civilisation'. Each test consisted of a sequence of 100 games played.

For the first game in each test, the dynamic civilisation started with rulebases with all weights equal. The dynamic civilisation was allowed to update the rulebases after each game. A game lasted until one of the civilisations was defeated, or until a certain period of time had elapsed. If a game ended due to the time restriction (which was rarely the case), the civilisation with the highest score was considered the winner of the game.

Games were played on two different maps, a small map and a large map. Games on a small map are usually decided swiftly, with fierce battles between weak armies. A large map allows for a slower-paced game, with long-lasting battles between strong armies. The two maps are discussed in detail in Appendix C, Section C.1.

Four different manually-designed game-AI variations, or 'tactics', were used for the static civilisation, namely the following.

Small Balanced Tactic: A 'balanced' tactic keeps a balance between offensive actions, defensive actions, and research. It is effective against many different playing styles employed by humans. The 'small balanced tactic' is employed on the small map.

Large Balanced Tactic: The 'large balanced tactic' is similar to the 'small balanced tactic', but is employed on the large map.

Soldier Rush: The 'soldier rush' aims at overwhelming the opponent with cheap offensive units in an early state of the game. Since the 'soldier rush' is most effective in fast games, it is employed on the small map.

Knight Rush: The 'knight rush' aims at quick technological advancement, launching large offences as soon as strong units are available. Since the 'knight rush' works best in slower-paced games, it is employed on the large map.

Details of the tactics are listed in Appendix C, Section C.3.

To quantify the relative performance of the dynamic civilisation against the static civilisation, the notion of the 'turning point' is defined as follows. After each game, an approximate randomisation test (Cohen, 1995) is performed using the overall fitness values over the most recent ten games, with the null hypothesis that both civilisations are equally strong. The dynamic civilisation is said to outperform the static civilisation if the randomisation test concludes that the null hypothesis can be rejected with a probability of 90%, in favour of the dynamic civilisation being stronger. The 'turning point' is the number of the first game in which the dynamic civilisation outperforms the static civilisation. Low values for the turning points indicate good efficiency of dynamic scripting.

6.2.4 Evaluation Results

The results of the evaluation of dynamic scripting in WARGUS are displayed in Table 6.1. From left to right, the table columns represent (i) the tactic used by the static civilisation, (ii) the number of tests, (iii) the average turning point, (iv) the median turning point, (v) the lowest turning point, (vi) the highest turning point, (vii) the number of tests that did not find a turning point within 100 games played, and (viii) the average number of games won during the test.

From the low values for the turning points for the two 'balanced' tactics, it may be concluded that the dynamic civilisation adapts effectively and efficiently. Therefore, dynamic scripting can be applied successfully to RTS games. However, the dynamic civilisation was unable to adapt to the two 'rush' tactics within 100 games. The reason for the inferior performance of the dynamic civilisation against the two 'rush' tactics is twofold, namely (i) the 'rush' tactics are optimised, in the sense that it is quite hard to design game AI that is able to deal with them, and (ii) the rulebase does not contain the appropriate knowledge to easily design game AI that is able to deal with the 'rush' tactics.

Note that this does not mean that dynamic scripting cannot use the rulebases to design an answer to the rush tactics. It can, and does so occasionally. However, the

Tactic	Tests	Average	Median	Lowest	Highest	> 100	Won
Small balanced	31	50	39	18	99	0	59.3
Large balanced	21	49	47	19	79	0	60.2
Soldier rush	10					10	1.2
Knight rush	10					10	2.3

Table 6.1: Evaluation results of dynamic scripting in WARGUS.

rulebases generate such an answer only on rare occasions. Therefore, it takes quite a long time before the rules of which such an answer consists have weights that are sufficiently high so that the answer occurs regularly. The requirement of efficiency disallows such a long learning time.

Perhaps not surprisingly, against the 'balanced' tactics, in some of the tests dynamic scripting encouraged the rulebases to create scripts that were very similar to the 'rush' tactics. Therefore, even if the 'rush' tactics had not been implemented manually, they would have been discovered automatically by dynamic scripting.

6.3 Evolutionary Tactics

The second step in combining online and offline adaptive game AI, is to use offline adaptive game AI to discover new tactics that can deal with the best results found by online adaptive game AI, and with the manually-designed tactics that online adaptive game AI was unable to deal with. In Section 4.1, offline evolutionary learning was used to design neural-network-based game AI for a strategy game. It was concluded that offline evolutionary learning is capable of evolving successful game AI, but that a neural network is not a suitable structure to store game AI. In the present section, a similar approach as in Section 4.1 is used to evolve script-based game AI. The goal is to design game AI for WARGUS, that has the ability to deal successfully with the two 'rush' tactics discussed in Section 6.2, which were difficult for dynamic scripting to deal with. This section discusses the experimental procedure used (6.3.1), the chromosome encoding (6.3.2), the fitness function used by the evolutionary algorithm (6.3.3), the genetic operators (6.3.4), the results achieved against the two 'rush' tactics (6.3.5), and a qualitative examination of the discovered solutions (6.3.6).

6.3.1 Experimental Procedure

An evolutionary algorithm was designed to evolve new tactics to be used in the WARGUS environment against a static civilisation using either the 'soldier rush' or the 'knight rush' tactic. The evolutionary algorithm used a population of size 50. The population was initialised with random (but legal) chromosomes. To select parent chromosomes for breeding, size-3 tournament selection was used (Goldberg, 1989).

Newly generated chromosomes replaced existing chromosomes in the population, using size-3 crowding (Goldberg, 1989).

The evolution continued until one of two stop criteria was fulfilled, namely (i) the fitness-stop criterion, or (ii) the run-stop criterion. The fitness-stop criterion aborts the evolution process when a chromosome with a target fitness value has been created. During preliminary experiments suitable target fitness values were determined, namely 0.75 against the 'soldier rush', and 0.70 against the 'knight rush'. The run-stop criterion aborts the evolution process when a maximum number of generations has been produced.

During preliminary experiments it was found that a maximum of only five generations (i.e., 250 new chromosomes) was sufficient to evolve successful game AI. When the evolution process ends, the chromosome with the highest fitness is considered the solution.

6.3.2 Encoding of Tactics

The evolutionary algorithm works with a population of chromosomes. In the present context, a chromosome represents a game-AI script. To encode a game-AI script for WARGUS, each gene in the chromosome represents one rule.

Four different gene types are distinguished, corresponding to the four basic rule categories mentioned in Subsection 6.2.2, namely (i) build genes, (ii) research genes, (iii) economy genes, and (iv) combat genes. Each gene consists of a 'rule ID' that indicates the type of gene ('B', 'R', 'E' and 'C', respectively), followed by values for the parameters needed by the gene.[6] The genes are grouped by states, and the start of a state is indicated by a separate marker ('S'), followed by the state number. Rule details can be found in Appendix C, Section C.4.

The chromosome design is illustrated in Figure 6.3. A schematic representation of the chromosome, divided into states, is shown at the top. Below it, a schematic representation of one state in the chromosome is shown, consisting of a state marker and a series of rule genes. Rule genes are identified by the number of the state for which they occur, followed by a period, followed by a sequence number. Below the state representation, a schematic representation of one rule is shown. At the bottom, part of an example chromosome is shown.[7]

Chromosomes for the initial population are generated randomly. The generating mechanism starts by randomly producing genes for the first state, allowing only genes that are legal in this state. When a build gene is produced that spawns a state transition, the generating mechanism switches to producing genes for the new state. This continues until the last state is reached, for which five genes are produced,

[6] Of the combat gene, there are actually twenty variations, one for each possible state. Each variation uses different parameters. They use rule ID's marked 'C1' to 'C20'.

[7] The example chromosome translates as follows. In state 1, first a defensive army is created with number 2, consisting of five soldiers. Then building type 4 is constructed. The construction of this building spawns a transition to state 3 (thus, from Figure 6.2 it can be derived that building type 4 is a 'blacksmith'). In state 3, first economy action 8 is executed, which is followed by research action 15. Finally, building type 3 (a 'lumbermill') is constructed, which spawns a transition to state 6.

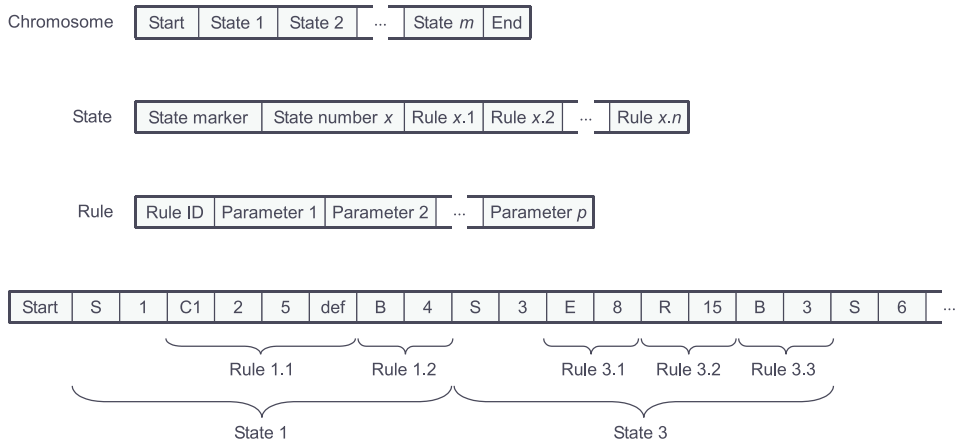

Figure 6.3: Chromosome design to store game AI for WARGUS.

and to which a loop is attached that continuously attacks with strong units. Thus it is ensured that only legal game-AI scripts are created.

6.3.3 Fitness Function

To determine the fitness of a chromosome, the chromosome is translated to a game-AI script. The game-AI script controls a dynamic civilisation against a static civilisation. A fitness function F measures the relative success of the game-AI script represented by the chromosome. Fitness function F for the dynamic player d, yielding a value in the range $[0, 1]$, is defined as follows.

$$F = \begin{cases} \min\left(\frac{C_T}{C_{max}} \cdot \frac{M_d}{M_d + M_c}, b\right) & \{d \text{ lost}\} \\ \max\left(\frac{M_d}{M_d + M_c}, b\right) & \{d \text{ won}\} \end{cases} \tag{6.5}$$

In this equation, C_T represents the timestep at which the game was finished (i.e., lost by one of the players, or aborted because time ran out), C_{max} represents the maximum timestep the game is allowed to continue to, M_d represents the 'military points' for the dynamic player, M_c represents the 'military points' for the dynamic player's opponent, and b is the break-even point. When a game is aborted because time ran out, the highest scoring civilisation wins (as calculated by equation 6.3). The factor $\frac{C_T}{C_{max}}$ ensures that a game AI that loses after a long game, is awarded a higher fitness than a game AI that loses after a short game.

Since WARGUS is completely deterministic, the fitness does not change if multiple

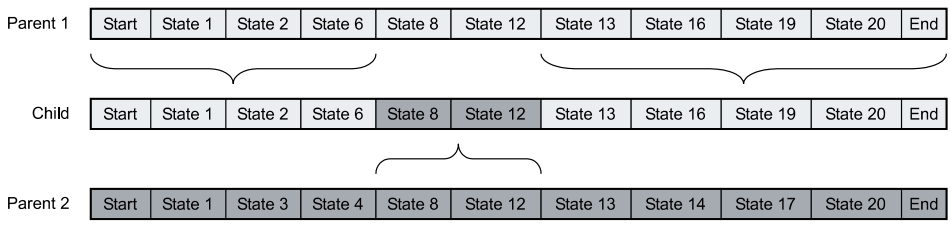

Figure 6.4: State crossover.

games are played. Were this not the case, the fitness would have been determined by playing several games and averaging over the fitness values per game.

6.3.4 Genetic Operators

To breed new chromosomes, four genetic operators were implemented. By design, all four genetic operators ensure that a child chromosome always represents a 'legal' game-AI script. Parent chromosomes are selected with a chance corresponding to their fitness values.

The genetic operators take into account 'activated' genes. An activated gene is a gene that represents a rule that was executed during the fitness determination. If a genetic operator produces a child chromosome that is equal to a parent chromosome for all activated genes, the child is rejected and a new child is generated. The reason is that genes that are not activated, are irrelevant to the game-AI script the chromosome represents.

The four genetic operators are the following.

- *State Crossover* selects two parent chromosomes, and copies states from either parent to the child chromosome. The genetic operator is controlled by 'matching states'. A 'matching state' is a state that exists in both parent chromosomes. Figure 6.2 makes evident that, for WARGUS, there are always at least four matching states, namely state 1, state 12, state 13, and state 20. State crossover will only be used when there are least three matching states with activated genes. A child chromosome is created as follows. States are copied from the first parent chromosome to the child chromosome, starting at state 1 and working down the chromosome. When there is a state transition to a matching state, there is a 50 per cent probability that from that point on, the role of the two parents is switched, and states are copied from the second parent. When the next state transition to a matching state is encountered, again a switch between the parents can occur. This continues until the last state has been copied. The process is illustrated in Figure 6.4. In the figure, parent switches occur at state 8 and at state 13.

- *Rule Replacement Mutation* copies one parent chromosome to a child chromosome. Then, all activated research, economy, and combat genes have a 25 per

Tactic	Average	Lowest	Highest	> 250
Soldier rush	0.78	0.73	0.85	2
Knight rush	0.75	0.71	0.84	0

Table 6.2: Evolutionary game AI in WARGUS results.

cent probability to be replaced with a random different economy, research, or combat gene. It is allowed to replace a gene of a certain type by a gene of a different gene type (e.g., it is allowed to replace a research gene by a combat gene). Build genes are excluded both for and as replacements, because they can spawn a state transition, which might corrupt the chromosome.

- *Biased Rule Mutation* copies one parent chromosome to a child chromosome. Then, all parameters for economy and combat genes have a 50 per cent probability to be mutated. Mutation changes the parameter value by adding a random integer value in the range $[-5, 5]$.
- *Randomisation* generates a random new child chromosome.

For each new child chromosome that is generated, randomisation has a 10 per cent probability to be selected, and the other three genetic operators each have a 30 per cent probability to be selected.

6.3.5 Evolutionary-Tactics Results

As a remedy against each of the two 'rush' tactics, ten tests were performed that generated a counter-tactic by evolutionary means. The results of the two series of ten tests are listed in Table 6.2. From left to right, the columns of the table represent (i) the tactic used by the static civilisation, (ii) the average of the solution-fitness values, (iii) the lowest solution-fitness value, (iv) the highest solution-fitness value, and (v) the number of tests that ended on the run-stop criterion. The table shows surprisingly high average, highest, and even lowest solution-fitness values. Therefore, it may be concluded that offline adaptive game AI was successful in rapidly discovering game-AI scripts able to defeat both 'rush' tactics used by the static civilisation.

6.3.6 Evolutionary-Tactics Discussion

About the solutions evolved against the 'soldier rush' tactic, the following observations were made. The 'soldier rush' is used on a small map. As is usual for a small map, the game played by the solutions was always short. Most solutions included only two states with activated genes. Basically, all ten solutions counter the 'soldier rush' tactic with a 'soldier rush' tactic of their own. In eight out of ten solutions, the solutions included building a 'blacksmith' very early in the game. Then, the

solutions selected at least two out of the three possible research advancements, after which large attack forces were created. These eight solutions succeed because they ensure their soldiers are quickly upgraded to be very effective, before they attack. The remaining two solutions overwhelmed the static civilisation with sheer numbers.

About the solutions evolved against the 'knight rush', the following observations were made. The 'knight rush' is used on a large map, which enticed longer games. On average, for each solution five or six states were activated. Against the 'knight rush', all solutions included training large number of 'workers' to be able to expand the civilisation quickly, and boosting the economy by exploiting additional resource sites after setting up defenses. Almost all solutions worked towards the goal of quickly creating advanced military units, in particular 'knights'. Seven out of ten solutions achieved this goal by employing a specific building order, namely a 'blacksmith', followed by a 'lumbermill', followed by a 'keep', followed by 'stables'. Two out of ten solutions preferred a building order that reached state 11 as fast as possible. State 11 is the first state that allows the building of the 'knights'.

Surprisingly, in several solutions against the 'knight rush', the game AI employed many 'catapults'. WARCRAFT II strategy guides generally consider 'catapults' to be inferior military units, because of their high costs and considerable vulnerability. A possible explanation for the successful use of 'catapults' by the evolutionary game AI is that, with their high damaging abilities and large range, they are particularly effective against tightly packed armies, such as groups of 'knights'.

6.4 Improving Online Adaptive Game AI

The third step in combining online and offline adaptive game AI, is to use the results achieved with offline adaptive game AI to improve the domain knowledge employed by online adaptive game AI. In Section 6.2, it was discovered that dynamic scripting did not achieve satisfying results against the two 'rush' tactics. Section 6.3 describes the evolution of new game-AI scripts, which are able to defeat the two 'rush' tactics. The present section discusses how the evolved game-AI scripts can be used to increase the reliability of dynamic scripting by improving the rulebases. Subsection 6.4.1 discusses how the evolved game-AI scripts are translated into rulebase improvements. Subsection 6.4.2 evaluates the new rulebases by repeating the experiment described in Section 6.2. Subsection 6.4.3 discusses the evaluation results.

6.4.1 Improving the Rulebases

Subsection 6.3.6 describes typical characteristics of the solutions discovered by the evolutionary game AI. The observations were used to manually create four new rules for the dynamic-scripting rulebases.

- Eight out of ten solutions against the 'soldier rush' contained a specific pattern of building and research, namely first building a 'blacksmith', then researching better weaponry and armour, followed by the creation of large offensive forces. A new rule was created that contained exactly this pattern.

- Against the 'knight rush', almost all solutions aimed at creating advanced military units quickly. This was acknowledged by creating a new rule, that checks whether it is possible to reach a state that allows the creation of advanced military units, by constructing one new building. If this is possible, the rule constructs that building, and creates an offensive force consisting of the advanced military units.

- Against the 'knight rush', all solutions included boosting the economy by constructing a new 'townhall'. The original rulebases, used in Section 6.2, contained rules for constructing a 'townhall', but these were invariably assigned low weights. The explanation is that a new 'townhall' is easily destroyed, and thus can only be successful if it can be defended against enemy interference. The solutions acknowledged this by first building up defenses. A new rule was created that combined the building of a defensive army, followed by the construction of a new 'townhall'.

- The best solution found against the 'knight rush' was translated into a new rule without interpretation. All activated genes for each state were translated and combined in one rule, and stored in the corresponding rulebase.

To keep the total number of rules constant, the new rules replaced existing rules. The replaced rules were rules that dealt with air combat. In the experiment described in Section 6.2, the air-combat rules always ended up with low weights.

Besides the creation of the four new rules, small changes were made to the existing combat rules, changing their parameters to increase the number of units of types preferred by the solutions, and to decrease the number of units of types avoided by the solutions. Through these changes, the use of 'catapults' was encouraged.

Details of the improved rulebase are supplied in Appendix C, Subsection C.5.2.

6.4.2 Evaluation of the Improved Rulebases

The experiment described in Section 6.2 was repeated, with dynamic scripting employing the improved rulebases. To encourage high weights, the maximum reward R_{max} and the maximum penalty P_{max} were both set to 400. The change of the maximum reward and penalty has little impact on the results achieved with dynamic scripting, since the weight values are compared to each other – it is not the absolute value of a weight that is important, but the value of a weight relative to competing weight values. However, with the higher values for R_{max} and P_{max}, the boundaries set to the weight values, W_{min} and W_{max}, are reached faster.

Table 6.3 summarises the achieved results. The columns represent the same variables as in Table 6.1. A comparison of Table 6.1 and Table 6.3 shows that the performance of dynamic scripting is considerably improved with the new rulebases. Against the two 'balanced' tactics, the average turning point is reduced by more than 50 per cent. Against the two 'rush' tactics, the number of games won out of 100 has increased considerably. It was observed that dynamic scripting assigned large weights to all four new rules, created in Subsection 6.4.1. Therefore, it may be

Tactic	Tests	Average	Median	Lowest	Highest	> 100	Won
Small balanced	11	19	14	10	34	0	72.5
Large balanced	11	24	26	10	61	0	66.4
Soldier rush	10					10	27.5
Knight rush	10					10	10.1

Table 6.3: Evaluation results of dynamic scripting in WARGUS, using improved rulebases.

concluded that the new rules are effective, and are the likely cause for the improved performance. The improved performance against all tactics indicates an improved reliability of dynamic scripting with the new rulebases, compared to dynamic scripting with the original rulebases.

6.4.3 Discussion

Despite the improvement of the reliability of dynamic scripting effectuated by the new rulebases, dynamic scripting is still unable to outperform the two 'rush' tactics statistically. The explanation of this fact is as follows. The two 'rush' tactics are 'super-tactics', that can only be defeated by very specific counter-tactics, with little room for variation. By design, dynamic scripting generates a variety of tactics at all times. Therefore, it is unlikely to make the appropriate choices enough times in a row to reach the turning point.

As was noted in Subsection 5.5.6, the fact that such super-tactics as the 'rush' tactics are possible at all, can be considered a weakness of the game design.[8] Adaptive game AI may be able to deal with super-tactics, if it is able recognise that a super-tactic is used, and has a pre-programmed 'answer' stored which it can use without activating a learning mechanism. However, a better solution would be to change the game design to make super-tactics impossible. If adaptive game AI is used during the 'quality assurance' phase of game development, super-tactics can be discovered before a game is released to the public, when there is still time to improve the game design.

One might wonder whether using counter-tactics against super-tactics to improve the domain knowledge stored in rulebases, may lead to the rulebases overfitting against the super-tactics. Since the experiment improved the performance of dynamic scripting not only against the 'rush' tactics, but also against the 'balanced' tactics, it seems overfitting has been avoided.

Actually, there is a good reason why the proposed procedure to improve the rulebases manages to avoid overfitting. The reason is a consequence of the principle discussed in Chapter 3, that solutions to hard instances encompass characteristics

[8] This is not to suggest that WARCRAFT II, on which WARGUS is based, has a weak game design. WARCRAFT II is a classic game that has gained lasting respect. However, 'rush' tactics are possible in the game, and can be considered detrimental to the game's entertainment value.

of solutions to easy instances. The 'rush' tactics can be considered hard instances, the 'balanced' tactics easy instances. The new rules derived from observing the solutions (i.e., the evolved counter-tactics) to the 'rush' tactics, implement typical characteristics of the solutions to the 'rush' tactics. These characteristics are likely to be able to deal successfully with easier tactics, too. Furthermore, as long as the new rules are *added* to a rulebase that can deal with easy tactics, or replace rules that are inferior anyway, then at worst the new rules are inconsequential against easy tactics. Therefore, overfitting is unlikely to occur.

To improve the domain knowledge for online adaptive game AI, the procedure prescribes extracting typical characteristics of offline evolved tactics. This step requires understanding and interpretation of the evolved tactics, which are activities that are difficult to perform automatically. Therefore, in the experiment the extraction was done manually. However, to some extent it should be possible to automate the extraction of new rules, especially since the effectiveness of the new rules can be tested by running the procedure again. This will be investigated in future work.

6.5 Acceptance

Offline adaptation of game AI, when applied before the game is released, is without risk. Therefore, game developers will not hesitate to apply offline adaptation if they consider the possible advantages it will bring worthwhile. In contrast, game developers will regard online adaptation of game AI with considerable suspicion. Since online adaptation of game AI can be used during playtesting to help improving static game AI, they might consider using online adaptation during the 'quality assurance' phase, as a first step on the road to include it in a released game.

I expect that, before game developers take a decision with regard to experimenting with online adaptive game AI, they will need some guarantee that the techniques discussed in this research generalise to their games. Three issues with regard to the generalisation of adaptive game AI are discussed below, namely (i) to what extent adaptive game AI generalises over the course of a game (6.5.1), (ii) to what extent adaptive game AI generalises to different game types (6.5.2), and (iii) to what extent the adaptive techniques generalise to different functionalities (6.5.3). A major issue for the acceptance of adaptive game-AI techniques is whether they contribute to the entertainment experienced by the human player of a game. This is discussed in Subsection 6.5.4. Finally, Subsection 6.5.5 discusses the future of adaptive game AI.

6.5.1 Generalisation over the Course of a Game

In the experiments described in Chapter 5 and 6, the adaptation techniques are tested against a static game AI in an effectively unchanging situation. In contrast, in modern games situations encountered by human players change over the course of the game. In general, the agents controlled by the human player will become more powerful when the game progresses. At the same time, the computer-controlled agents that oppose the human player will become more powerful too. The question

is warranted whether adaptive game AI can be expected to perform well in these changing circumstances.

The answer is that it depends on the design of the domain knowledge (e.g., the dynamic-scripting rulebases) employed by the adaptive game AI. Adaptive game AI can be expected to function well over the course of the game, if the domain knowledge is formulated sufficiently general to describe rules and facts that hold for most situations in the game. As Appendix A shows, the rulebases designed for the CRPG simulation described in Chapter 5 are not sufficiently general. For instance, a rule that casts a 'Fireball' spell works fine as long as the 'Fireball' spell is a good spell to use, but fails when there are better alternatives available. Contrariwise, as Appendix B shows, the rulebases designed for NEVERWINTER NIGHTS only refer to actions in a general manner, taking into account the current status of the game.

Of course, to achieve a generalised implementation of game AI, the game should allow generalised domain knowledge to be formulated. For instance, a rule stating that 'an effective action against a group of enemies standing close together is attacking them with an area-effect weapon' should hold for the whole course of the game, otherwise it does not reflect correct domain knowledge. However, even for games where it is difficult to formulate domain knowledge in general, adaptive game AI can be implemented by using different rulebases for different game states. In the present chapter, this approach has been used, with great success, to deal with the changing circumstances over the course of an RTS game.

6.5.2 Generalisation to Different Game Types

To what extent can the techniques for adaptive game AI, discussed in this thesis, be used in different games types?

For offline adaptive game AI, there are no real restrictions to game types, since offline adaptive game AI can generate literally anything. A major obstruction to using offline adaptive game AI is that offline learning techniques can take a huge amount of computational resources before results are achieved. Usually, the amount of required computational resources can be kept relatively small by carefully designing and implementing the offline adaptive game AI. However, careful design and implementation require a considerable, and thus expensive, investment on the part of the game developers. Therefore, offline adaptive game AI should be applied to games where it can be really worthwhile. Typically, these are games with complex game AI, such as CRPGs and strategy games.

For online adaptive game AI, dynamic scripting has already been shown applicable to two completely different types of games with highly complex game AI, namely CRPGs (Chapter 5) and RTS games (the present chapter; furthermore, Dahlbom (2004) offers an alternative implementation of dynamic scripting in RTS games). By extrapolation, dynamic scripting is also applicable to different game types, that use scripted game AI with a complexity less than CRPGs and RTS games. This is the majority of games on the market today.

To games that use game AI not implemented in scripts, dynamic scripting is not directly applicable. However, based on the idea that domain knowledge must be the

core of an online adaptive game-AI technique, an alternative for dynamic scripting may be designed. For instance, if game AI is based on a finite-state machine, state transitions can be extracted from a rulebase to construct the finite-state machine, in a way similar to dynamic scripting's selection of rules for a game-AI script.

6.5.3 Generalisation of Functions

In the research discussed in this thesis, game AI is developed with as its main function competing with the human player. However, the investigated techniques are not restricted to that function.

Obviously, offline adaptive game AI, investigated in Chapter 4 and in the present chapter, is based on evolutionary learning, which can be applied to many different problem domains (cf. Goldberg, 1989; Davis, 1991; Michalewicz, 1992). For evolutionary learning, the only requirement for use is that an adequate fitness function can be designed (Goldberg, 1989).

Online adaptive game AI in the form of dynamic scripting, investigated in Chapter 5 and in the present chapter, can be applied to any function that meets three requirements (as mentioned before in Subsection 5.1.1): (i) the function can be scripted, (ii) domain knowledge on the characteristics of a successful function can be collected, and (iii) an evaluation function can be designed to assess how successful the function was executed. Such functions are not only found in games, but also in less 'frivolous' application areas, such as multi-agent systems.

6.5.4 Learning to Entertain

The main goal of a game is to provide entertainment. If online adaptive game AI is not beneficial to the entertainment experienced by human players, game developers will not be interested in implementing it. Therefore, the question is warranted whether online adaptive game AI really improves a game's entertainment factor.

It is evident that not every human player is entertained by the same aspects of a game. Charles and Livingstone (2004) differentiate between players that desire to master a game, and players that desire to experience variety in a game. Obviously, the first group of players will not enjoy adaptive game AI, since the game will adapt when players are getting close to mastering it. However, the second group of players will enjoy the variety adaptive game AI provides.

How can be assessed whether the techniques discussed in this thesis, in particular dynamic scripting, improves the entertainment of a game, for at least those players that enjoy the variety and the increased challenge? An answer to this question may be discovered by a large-scale psychological investigation of players of a game that can be experienced with or without adaptive game AI. However, such an investigation is beyond the scope of this thesis. Still, literature provides indications that adaptive game AI improves the entertainment of games, as explained below.

Most players are intrinsically motivated to play a game, i.e., they are not forced to play the game, but do so purely for pleasure. Empirical studies have linked intrinsic motivation to the concept of 'presence' (also referred to as 'immersion'

or 'suspension of disbelief'); the stronger the sense of 'presence', the higher the intrinsic motivation, and thus the greater the entertainment experienced (Heeter, 1992; IJsselsteijn *et al.*, 2004). Since adaptive game AI allows computer-controlled agents to avoid the continuous repetition of mistakes, it improves the feeling of immersion experienced by the human player, and thus contributes positively to the entertainment provided by the game.

To measure the entertainment provided by analytical games, Iida and Yoshimura (2003) formulated a theory of game refinement. According to the theory, game refinement is expressed by the formula $\frac{\sqrt{B}}{D}$, where B represents the branching factor of the game, and D represents the game depth, i.e., the average number of moves in the game until the outcome is decided. Game refinement was calculated for several CHESS variations (Iida, Takeshita, and Yoshimura, 2002) and for the game of MAH JONG (Iida *et al.*, 2004). Iida *et al.* (2002) surmised that for optimal entertainment, the refinement value of a game must be in the neighbourhood of 0.07.

Unfortunately, the refinement formula cannot be easily translated from analytical games to commercial games, since the branching factor for commercial games is very difficult to determine.[9] It seems clear that, in order to apply the refinement formula to commercial games, theory must be developed to determine how the concepts of 'branching factor' and 'game depth' can be translated to commercial games. Yannakakis and Hallam (2004) proposed a metric to measure the 'interest value' of commercial predator-prey games (where the human player is the 'prey'), based on the prey's 'lifetime', and the predator's 'diversity in tactics'. However, their metric might be criticised for the fact that it equates increased lifetime for the human player with increased entertainment value, while it seems evident that humans are not entertained by a game that drags on endlessly.

Even though the refinement formula cannot be applied to games directly, the basis for the theory of refinement is applicable to all games. Iida and Yoshimura (2003) derive the theory of refinement from the observation that the entertainment experienced from a game results from three essential properties of games, namely (i) complexity, (ii) fairness, and (iii) refinement.

Complexity is translated as 'noble uncertainty', i.e., to be entertaining, the rules of the game must be of sufficient complexity that players feel that it is possible (and useful) to discover new, more advanced tactics. In commercial games, against inferior game AI, there is no need to design new tactics. Adaptive game AI has the ability to increase the playing strength of computer-controlled agents, and thus stimulates complexity.

Fairness is translated as 'draw ratio', i.e., the better two opponents are matched, the higher the entertainment they will experience. Static game AI always plays a game with the same level of skill, and thus is likely to play the game significantly

[9] For example, in a CRPG, a wizard may have spells that can be unleashed to any location within range. Use of such a spell cannot be considered just one possible move, since the spell effect depends on its target location. However, use of such a spell also cannot be considered a virtually endless number of moves, since the practical number of useful locations will be limited. Still, for most complex commercial games the branching factor will be much higher than the branching factor for most analytical games.

worse than human players.[10] To compensate for inferior game AI, game developers will often supply computer-controlled agents with 'physical' attributes that outrank human-controlled agents. Such design detracts from the fairness of matching the physical aspects of the agents controlled by the human player and the computer. Adaptive game AI has the ability to improve the playing strength of computer-controlled agents against a human player, even when the physical attributes of the computer-controlled agents are equal to those of the agents controlled by the human player. Thus, adaptive game AI stimulates fairness.

Refinement is translated as the 'seesaw game', i.e., the optimal length of time for which the outcome of the game is uncertain. Entertainment is high if the game is not decided 'too fast', and does not drag on after the outcome has been decided. In this respect, adaptive game AI increases the period of time needed for a human player to master a game. Furthermore, when adaptive game AI is enhanced with difficulty scaling, it will also ensure that novice players experience a well-matched game. Thus, adaptive game AI stimulates refinement.

In conclusion, adaptive game AI has a beneficial effect on all aspects which form the basis of the theory of refinement. Therefore, as far as the theory of game refinement is applicable to commercial games, the entertainment provided by commercial games benefits from adaptive game AI.

6.5.5 The Future of Adaptive Game AI

Observing the state of the art in games today, it is clear that game AI has a long road to travel before truly believable computer-controlled characters are implemented. The ability to correct mistakes (self-correction), and the ability to adapt to changing circumstances (creativity), are essential elements of a believable character. Despite this, the consensus amongst game developers and publishers seems to be that adaptive game AI is something to be avoided. Their distrust stems not so much from a lack of interest, but more from laziness (Rabin, 2004b) and a fear of breaking game AI that more or less worked when designed manually (Woodcock, 2002). However, as soon as one company manages to pull off adaptive game AI successfully, the others are forced to join in, lest they will be unable to compete.

Dynamic scripting has been shown to be able to implement successful online adaptive game AI, proving that online adaptive game AI is possible in state-of-the-art games. The question is therefore not if, but when adaptive game AI will become a standard element of games.

6.6 Chapter Summary

This chapter discussed how adaptive game AI can be applied in practice. Offline adaptive game AI can be used during the 'quality assurance' phase of game develop-

[10] One might assume that it is also possible for static game AI to play the game better than human players, but human players that lose a game too often will, in general, quit playing (Livingstone and Charles, 2004).

ment to fine-tune and improve manually-designed game AI. Online adaptive game AI allows the game AI to adapt to human-player tactics after a game has been released. Since game developers consider online adaptive game AI risky, during the 'quality assurance' phase the reliability of the game AI must be ensured by confirming that it meets the requirements specified in Subsection 2.3.4.

To increase the reliability of online adaptive game AI, offline adaptive game AI can be used to improve the domain knowledge used by online adaptive game AI. A three-step procedure is proposed to effectuate this, namely (i) using online adaptive game AI to discover strong tactics, (ii) using offline adaptive game AI to evolve counter-tactics against the discovered tactics, and against manually-designed strong tactics, and (iii) extracting characteristics from the evolved counter-tactics to add to the domain knowledge used by the online adaptive game AI. The procedure was empirically validated by applying it to dynamic scripting in a Real-Time Strategy (RTS) game.

The chapter also discussed several generalisation issues of adaptive game AI. It was argued that the techniques discussed in this thesis generalise over the course of a game, and to different game types. The techniques are not limited to game AI that competes with human players, but can be applied to other functionalities in games, and in other applications as well. Finally, it was argued that adaptive game AI will contribute to the entertainment experienced by human players of a game, and that, in the future, adaptive game AI will become a standard element of games.

Chapter 7

Conclusion

> The real danger is not that computers will begin to think like men,
> but that men will begin to think like computers.
> — Sydney J. Harris (1917–1986).

This chapter provides a conclusive answer to the problem statement and research questions posed in Chapter 1. Section 7.1 restates and answers the four research questions. Section 7.2 translates the answers to the research questions to an answer to the problem statement. Section 7.3 looks at future work. The chapter ends with concluding remarks in Section 7.4.

7.1 Answer to Research Questions

The four research questions, stated in Section 1.5, are answered in the present section. Subsection 7.1.1 answers the first research question, on offline adaptive game AI. Subsection 7.1.2 answers the second research question, on online adaptive game AI. Subsection 7.1.3 answers the third research question, on difficulty scaling. Subsection 7.1.4 answers the fourth research question, on the integration of adaptive game AI in the game-development process.

7.1.1 Offline Adaptive Game AI

The first research question reads:

> **Research question 1**: To what extent can offline machine-learning techniques be used to increase the effectiveness of game AI?

The answer to the first research question is derived from Chapters 3, 4, and 6.

Chapter 3 discussed the creation of successful agent controllers with evolutionary learning. It showed that by 'doping' (or 'seeding') the initial population with a solution to a hard problem instance, evolved agent controllers are significantly more

effective than agent controllers evolved without doping. Since game AI that determines the behaviour of an in-game agent, is equivalent to an agent controller, it may be concluded that the application of offline machine-learning techniques to game AI will achieve more effective results if it concentrates on hard game situations first. As stated in Chapter 6, the beneficial effect of focussing on hard instances for deriving generalised game AI, is an explanation for the fact that overfitting is avoided when generalised game AI is improved by exploiting tactics used by game AI that is designed to defeat a superior opponent.

Chapter 4 discussed evolutionary game AI. It showed that offline evolutionary game AI is suitable for detecting possible exploits in manually-programmed game AI, and for discovering new tactics. It also indicated that, for offline evolutionary game AI, the use of a learning structure that is less suitable for storing game AI will negatively influence the success of the achieved results. Furthermore, it will negatively influence the efficiency by which results are generated. For game AI that is best stored in production rules, a learning structure should be used that is designed to evolve scripts. In Chapter 6, evolutionary game AI was used to evolve scripts, and proved to be not only successful, but also very efficient.

Chapter 6 discussed the application of offline evolutionary game AI in practice. The chapter described a three-step procedure to use offline evolutionary game AI to improve the domain knowledge used by online adaptive game AI during the 'quality assurance' phase of game development, thereby improving the reliability of online adaptive game AI. It showed that this application of offline adaptive game AI could be very successful. Since the computational requirements for adaptive game AI set no restrictions to offline adaptive game AI, the only limitations to the application of offline machine-learning techniques are available resources (i.e., time and money). Furthermore, the use of offline adaptive game AI during 'quality assurance' is essentially risk-free. Therefore, an application of offline adaptive game AI as described by the three-step procedure is likely to be successful in the practice of game development, and easily adopted by game developers.

In conclusion, the answer to the first research question is that:

- computational requirements form no obstacle for the application of offline machine-learning techniques to game AI;
- offline machine-learning techniques can increase the effectiveness of game AI by (i) detecting exploits, (ii) suggesting new tactics, and (iii) improving the domain knowledge used by online machine-learning techniques; and
- offline machine-learning techniques achieve superior results when designing effective game AI, when they concentrate on hard problem instances.

7.1.2 Online Adaptive Game AI

The second research question reads:

> **Research question 2**: To what extent can online machine-learning techniques be used to increase the effectiveness of game AI?

The answer to the second research question is derived from Chapters 2, 4, 5, and 6.

Chapter 2 listed four computational requirements (namely the requirements of speed, effectiveness, robustness, and efficiency) and four functional requirements (namely the requirements of clarity, variety, consistency, and scalability) for machine-learning techniques to adapt game AI online. When a technique meets the four computational requirements, it is able to increase the effectiveness of game AI. When it also meets the functional requirements of clarity, variety, and consistency, it is acceptable to game developers to increase the effectiveness of game AI online. It was also argued that any online machine-learning technique for improving the effectiveness of game AI is necessarily based on domain knowledge.

Chapter 4 discussed evolutionary game AI. It showed that online evolutionary game AI is able to increase the effectiveness of game AI during game-play. However, the success of online evolutionary game AI was shown to depend on the potential solutions residing in a small search space. In general, when evolving game AI that is complex, online evolutionary game AI will not meet the computational requirement of efficiency. Therefore, to adapt complex game AI online, a different approach needs to be used.

Chapter 5 presented 'dynamic scripting', an online machine-learning technique for game AI. Dynamic scripting was shown to meet all four computational requirements, and the functional requirements of clarity and variety. Furthermore, an outlier-reduction enhancement was presented for dynamic scripting, which allows it to meet the functional requirement of consistency. Therefore, dynamic scripting is a machine-learning technique suitable for increasing the effectiveness of game AI online.

The success of dynamic scripting heavily depends on the quality of the domain knowledge it uses (in the form of tactical rules). Chapter 6 shows how offline machine-learning techniques can be used to increase the quality of the domain knowledge used by dynamic scripting, thereby improving its reliability.

In conclusion, the answer to the second research question is that:

- online machine-learning techniques for game AI are heavily dependent on domain knowledge;
- online machine-learning techniques can improve the effectiveness of game AI, while meeting all requirements for acceptance; and
- offline machine-learning techniques can be used to improve the reliability of online adaptive game AI.

7.1.3 Difficulty Scaling

The third research question reads:

> **Research question 3**: To what extent can machine-learning techniques be used to scale the difficulty level of game AI to meet the human player's level of skill?

The answer to the third research question is derived from Chapter 5. The chapter presents dynamic scripting as a machine-learning technique for the online adaptation of game AI. Dynamic scripting was initially designed to increase the effectiveness of game AI. As the answer to the second research question indicates, this initial version of dynamic scripting did not meet the functional requirement of scalability. Thus, it could only be used to increase the effectiveness of game AI, not to match the playing strengths of the game AI and the human player.

A difficulty-scaling enhancement to dynamic scripting was presented that allows it to match automatically the playing strength of the game AI and the playing strength of the human player. Of the several possible implementations of a difficulty-scaling enhancement, 'top culling' was most successful, being reliable, easy to implement, and able to match the playing strength of both inferior and superior opponents.[1] Top culling functions by automatically making the most successful tactical domain knowledge unavailable when the game AI is detected to be too strong, and by automatically making it available again when the game AI is detected to be too weak. After applying top culling, dynamic scripting meets all four computational requirements and all four functional requirements.

In conclusion, the answer to the third research question is that online adaptive game AI can be made to scale its playing strength to meet the human player's level of skill, by changing automatically the availability of domain knowledge that realises the most effective game AI.

7.1.4 Integration in State-of-the-Art Games

The fourth research question reads:

> **Research question 4**: How can adaptive game AI be integrated in the game-development process of state-of-the-art games?

The answer to the fourth research question is derived from Chapters 5 and 6.

Chapter 5 presents dynamic scripting as a technique for online adaptive game AI. The chapter shows that dynamic scripting can be used in state-of-the-art games, by implementing it in the game NEVERWINTER NIGHTS (2002), and showing it to be successful. The chapter also argues that online adaptive game AI gives best results against human players that do not use highly-successful tactics, i.e., non-expert players.

Chapter 6 specifically discusses the integration of adaptive game AI in the development process of state-of-the-art games. For games that use only manually-designed game AI, offline adaptive game AI can be used before the game's release, during the 'quality assurance' phase of game development, for detecting possible exploits in the game AI, and for discovering new tactics. Since there is little risk associated with the use of offline adaptive game AI, game developers will not hesitate to use it when they feel it is worth their while.

[1] Of course, using difficulty scaling the game AI will never get more effective than the most effective results achieved with online adaptive game AI without a difficulty-scaling enhancement.

Since online adaptive game AI is still new to games, its inclusion must be considered during the earliest phases of game development. Game developers and publishers feel adaptive game AI is risky. Only when they are convinced that adaptive game AI is reliable (i.e., meets the requirements specified in Chapter 2), they will be willing to use it in released games. Offline adaptive game AI can be used to increase the reliability of online adaptive game AI, by improving the quality of the domain knowledge used.

In conclusion, the answer to the fourth research question is that:

- offline adaptive game AI can be used during the 'quality assurance' phase of game development to improve the quality of manually-designed game AI;
- online adaptive game AI can be used in released games when game developers and publishers are convinced of its reliability;
- the reliability of online adaptive game AI can be guaranteed by showing that it meets the four computational and four functional requirements; and
- the reliability of online adaptive game AI can be increased by using offline adaptive game AI to improve the quality of the domain knowledge used.

7.2 Answer to Problem Statement

The problem statement reads:

> **Problem statement**: To what extent can machine-learning techniques be used to increase the quality of complex game AI?

Taking into account the answers to the the research questions in Section 7.1, the answer to the problem statement is that:

- reliability of online adaptive game AI is guaranteed if it meets the four computational and four functional requirements;
- offline machine-learning techniques can be used during the 'quality assurance' phase of game development to increase the effectiveness of game AI by (i) detecting exploits, (ii) suggesting new tactics, and (iii) increasing the reliability of online adaptive game AI by improving the quality of the domain knowledge used;
- after a game's release, online machine-learning techniques can (i) improve the effectiveness of game AI, and (ii) scale the difficulty level of game AI to match the playing strength of the human player; and
- game developers and publishers will consider using online adaptive game AI when they are convinced that it is reliable.

7.3 Future Work

The research discussed in this thesis indicates three areas of future research.

1. *DECA Validation*: Chapter 3 presents the Doping-driven Evolutionary Control Algorithm (DECA). The characteristics of DECA require further investigation in future work. It must be determined for which tasks and under which conditions DECA performs better or worse than alternative techniques. In particular, in empirical studies DECA should be compared to hillclimbing (3.5.2), multitask learning (3.5.3), multi-objective learning (3.5.4), and boosting (3.5.5). In addition to these empirical studies, a solid explanation for the doping effect is required to identify problems to which DECA can be applied successfully. To this purpose, the key assumption in the explanation for the doping effect, namely the supposed asymmetry of the search space with respect to easy and hard solutions (3.1.3), needs verification. Furthermore, confirmation is needed for the belief that solutions to harder task instances encompassing characteristics of solutions to easier task instances underlies DECA's success (3.5.1). To this end, DECA should be tested on a variety of benchmark problems, designed to exhibit specific characteristics with respect to the structure of the search space. Tracing the lineage of the best evolved solutions back to the doped solutions will be a key activity in understanding the factors responsible for DECA's success.

2. *Entertainment Validation*: Chapter 1 stated that the goal of games is to provide entertainment. Entertainment is a subjective experience of human players. While this thesis argued that adaptive game AI is able to increase the entertainment value of games, it used only experiments wherein static game AI replaced the human player. In future work, an empirical study should investigate the effectiveness and entertainment value of online adaptive game AI (e.g., dynamic scripting) in games played against actual human players. While such a study requires many subjects and a careful experimental design, the game-play experiences of human players are important to convince game developers to adopt dynamic scripting in their games.

3. *Adaptive Game AI for Multi-player Games*: The adaptive game AI discussed in this thesis focussed on learning from a single human player. For future work, a logical extension is adaptive game AI that learns from multiple parallel players. A data store can be used to store samples of game-play experiences against multiple human players. Game AI can use the data store (i) to guide its decisions using a case-based reasoning approach, and (ii) as a model to predict the effect of actions which it deliberates. An approach to adaptive game AI based on a data store can achieve at least the same reliability as the adaptive game AI discussed in this thesis, and probably even a higher reliability. Moreover, it provides an approach to reduce the effect of non-determinism in games (since the number of samples increases with the number of human players), and to design completely new tactics online (since the data

store can be used as a model). Three problems that this research must deal with are (i) the design of a rapidly accessible data store that contains game-play samples and allows a relevant mapping of game-play situations to the stored samples, (ii) the design of an algorithm that uses the data store to allow game AI to respond to new game-play situations, and (iii) the design of an algorithm that uses the data store to allow game AI to match the playing strength of the human player, without affecting negatively the entertainment derived from the game.

7.4 Final Thoughts on Dynamic Scripting

A famous folk figure in the Arabic world is the Mullah Nasrudin. Nasrudin is a sage and a scoundrel, whose wisdom of words seems to be ever clouded by his reputation as a prankster. While some of the tales about Nasrudin are outright jokes, most have a deeper meaning that is intended to transfer philosophical thinking in an amusing package. One of the stories about Nasrudin, recanted by Shah (1968), goes as follows:

> Nasrudin stood up in the market-place and started to address the throng.
>
> "O people! Do you want knowledge without difficulties, truth without falsehood, attainment without effort, progress without sacrifice?"
>
> Very soon a large crowd gathered, everyone shouting: "Yes, yes!"
>
> "Excellent!" said the Mulla. "I only wanted to know. You may rely upon me to tell you all about it if I ever discover any such thing."

The meaning behind this story is evident: Nasrudin's appeal to the crowd lists four desirable features of progression, which the crowd would love to believe are possible, but which he feels are evidently unattainable regardless how much people covet them.

When I read this story, I noticed by how similar the four features which Nasrudin mentions are to the four computational requirements of online adaptive game AI, discussed in Section 2.3.4. 'Knowledge' can be interpreted as game AI, and so 'knowledge without difficulties' becomes the requirement of efficiency: quick, easy steps towards successful game AI. 'Truth' can be interpreted as correct domain knowledge, and so 'truth without falsehood' becomes the requirement of robustness: correct domain knowledge that does not get tainted by inferior domain knowledge. 'Attainment' can be interpreted as the discovery of successful game AI, and so 'attainment without effort' becomes the requirement of speed: the achievement of successful game AI without investing much in the name of resources. 'Progress' can be interpreted as the process of creating increasingly effective game AI, and so 'progress without sacrifice' becomes the requirement of effectiveness: continuous improvements of game AI without sacrificing intermediate results by installing game AI of inferior quality.

Nasrudin believes that the features are impossible to achieve, and the crowd, slightly embarrassed by its initial enthusiasm, will probably agree to that. Indeed, the features do sound too good to be true. Yet, for online adaptive game AI these features are requirements. And, as has been shown in this thesis, they actually are attainable.

When presenting some of the results discussed in this thesis at conferences, occasionally I have been confronted with the remark that the dynamic-scripting technique is rather simple. In these instances, the remark was meant to be criticising, as if something simple is somehow unworthy of scientific merit. I would like to point out, that I sincerely believe that it is precisely the simplicity of dynamic scripting that allows it to meet all four computational requirements. While more complex techniques may be designed, and may discover even more successful game AI, if they fail to meet the four computational requirements they are of no interest to game developers. In this thesis I sought the combination of scientific progress and practical applicability, and the mere fact that a successful approach to this combination lacks complexity is no reason to disqualify it.

Interestingly, when I first came up with the dynamic-scripting technique, I almost disqualified the technique myself, thinking "it is too easy" and "if it would work, surely someone else would have thought of it first". Much to my surprise, dynamic scripting worked better than I had expected. For me, the surprise has gone now, but what remains is the realisation that dynamic scripting is one of those techniques that are only obvious in hindsight.

References

Adamatzky, A. (2000). CREATURES - Artificial Life, Autonomous Agents and Gaming Environment. *Kybernetes: The International Journal of Systems & Cybernetics*, Vol. 29, No. 2.

Aha, D.W. and Molineaux, M. (2004). Integrating Learning in Interactive Gaming Simulators. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 49–53, AAAI Press, Menlo Park, CA.

Alba, E., Aldana, J.F., and Troya, J.M. (1993). Genetic Algorithms as Heuristics for Optimizing ANN Design. *Artificial Neural Nets and Genetic Algorithms* (eds. R.F. Albrecht, C.R. Reeves, and N.C. Steel), pp. 683–690, Springer-Verlag, Wien, Austria.

Albrecht, R.F., Reeves, C.R., and Steel, N.C. (1993). *Artificial Neural Nets and Genetic Algorithms*. Springer-Verlag, Wien, Austria.

Aleksander, I. and Morton, H. (1990). *An Introduction to Neural Computing*. Chapman and Hall, London, UK.

Allen, M.J., Suliman, H., Wen, Z., Gough, N.E., and Mehdi, Q.H. (2001). Directions for Future Game Development. *Proceedings of the Second International Conference on Intelligent Games and Simulation* (eds. Q. Mehdi, N. Gough, and D. Al-Dabass), pp. 22–32, SCS Europe Bvba, Ghent, Belgium.

Arkin, R.C. (1998). *Behaviour-Based Robotics*. MIT Press, Cambridge, MA.

Asada, M. and Kitano, H. (1999). The Robocup Challenge. *Robotics and Autonomous Systems*, Vol. 29, No. 1, pp. 3–12.

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, Oxford, UK.

Bakkes, S. (2003). *Learning to Play as a Team: Designing an Adaptive Mechanism for Team-Oriented Artificial Intelligence*. M.Sc. thesis. Universiteit Maastricht, Maastricht, The Netherlands.

Bakkes, S., Spronck, P.H.M., and Postma, E.O. (2004). TEAM: The Team-oriented Evolutionary Adaptability Mechanism. *Entertainment Computing – ICEC 2004* (ed. M. Rauterberg), Lecture Notes in Computer Science 3166, pp. 273–282, Springer-Verlag, Berlin, Germany.

Ballard, D. (1997). *An Introduction to Natural Computation.* MIT Press, Cambridge, MA.

Baratz, A. (2001). The Stage of the Game. *Ars Technica.* www.arstechnica.com/reviews/01q3/gaminghistory/ghistory-1.html.

Baxter, J., Tridgell, A., and Waever, L. (1998). Experiments in Parameter learning Using Temporal Differences. *ICCA Journal*, Vol. 21, No. 2, pp. 84–99.

Biasillo, G. (2002). Training an AI to Race. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 455–459, Charles River Media, Inc., Hingham, MA.

Bishop, C.M. (1995). *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, UK.

Braun, H. and Weisbrod, J. (1993). Evolving Neural Feedforward Networks. *Artificial Neural Nets and Genetic Algorithms* (eds. R.F. Albrecht, C.R. Reeves, and N.C. Steel), pp. 25–32, Springer-Verlag, Wien, Austria.

Brockington, M. and Darrah, M. (2002). How *Not* to Implement a Basic Scripting Language. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 548–554, Charles River Media, Inc., Hingham, MA.

Buro, M. (1997). The Othello Match of the Year: Takeshi Murakami vs. Logistello. *ICCA Journal*, Vol. 20, No. 3, pp. 189–193.

Buro, M. (2003a). ORTS: A Hack-Free RTS Game Environment. *Computers and Games: Third International Conference, CG 2002* (eds. J. Schaeffer, M. Müller, and Y. Björnsson), Vol. 2883 of *Lecture Notes in Computer Science*, pp. 280–291, Springer-Verlag, Heidelberg, Germany.

Buro, M. (2003b). RTS Games as Test-Bed for Real-Time AI Research. *Proceedings of the 7th Joint Conference on Information Science (JCIS 2003)* (eds. K. Chen, S-H. Chen, H-D. Cheng, D.K.Y. Chiu, S. Das, R. Duro, Z. Jiang, N. Kasabov, E. Kerre, H.V. Leong, Q. Li, M. Lu, M. Grana Romay, D. Ventura, P.P. Wang, and J. Wu), pp. 481–484.

Buro, M. (2004). Call for AI Research in RTS Games. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 139–142, AAAI Press, Menlo Park, CA.

Caruana, R. (1997). Multitask Learning. *Machine Learning*, Vol. 28, pp. 41–75.

Champandard, A.J. (2004). *AI Game Development.* New Riders, Indianapolis, IN.

Chan, B., Denzinger, J., Gates, D., Loose, K., and Buchanan, J. (2004). Evolutionary Behavior Testing of Commercial Computer Games. *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pp. 125–132, IEEE Press, Piscataway, NJ.

Charles, D. and Black, M. (2004). Dynamic Player Modelling: A Framework for Player-Centric Digital Games. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass), pp. 29–35, University of Wolverhampton, Wolverhampton, UK.

Charles, D. and Livingstone, D. (2004). AI: The Missing Link in Game Interface Design. *Entertainment Computing – ICEC 2004* (ed. M. Rauterberg), Lecture Notes in Computer Science 3166, pp. 351–354, Springer-Verlag, Berlin, Germany.

Charles, D. and McGlinchey, S. (2004). The Past, Present and Future of Artificial Neural Networks in Digital Games. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass), pp. 163–169, University of Wolverhampton, Wolverhampton, UK.

Clarke, A.C. (1968). *2001: A Space Odyssey*. Sidgwick and Jackson, Ltd, London, UK.

Cohen, P.R. (1995). *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, MA.

Cook, M., Tweet, J., and Williams, S. (2000). *Dungeons & Dragons Player's Handbook*. Wizards of the Coast, Renton, WA.

Crawford, C. (1984). *The Art of Computer Game Design*. McGraw-Hill. www.vancouver.wsu.edu/fac/peabody/game-book/Coverpage.html.

Dahlbom, A. (2004). *An Adaptive AI for Real-Time Strategy Games*. M.Sc. thesis. Högskolan i Skövde, Skövde, Sweden.

Darwin, C. (1859). *The Origin of Species by Means of Natural Selection: Or, the Preservation of Favored Races in the Struggle of Life*. John Murray Publishers, London, UK.

Davis, L. (1991). *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, NY.

Dawkins, R. (1976). *The Selfish Gene*. Oxford University Press, Oxford, UK.

Dawkins, R. (1986). *The Blind Watchmaker*. Penguin Books, London, UK.

Demasi, P. and Cruz, A.J. de O. (2002). Online Coevolution for Action Games. *International Journal of Intelligent Games and Simulation*, Vol. 2, No. 2, pp. 80–88.

Demasi, P. and Cruz, A.J. de O. (2003). Anticipating Opponent Behaviour Using Sequential Prediction and Real-Time Fuzzy Rule Learning. *Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)* (eds. Q. Mehdi, N. Gough, and S. Natkin), pp. 101–105, EUROSIS, Ghent, Belgium.

Donkers, H.H.L.M. (2003). *Nosce Hostem: Searching with Opponent Models*. Ph.D. thesis, Universiteit Maastricht. Universitaire Pers Maastricht, Maastricht, The Netherlands.

Dumitrescu, D., Lazzerini, B., Jain, L.C., and Dumitrescu, A. (2000). *Evolutionary Computation*. CRC Press, Boca Raton, FL.

Elman, J.L. (1990). Finding Structure in Time. *Cognitive Science*, Vol. 14, pp. 179–211.

Enzenberger, M. (2003). Evaluation in Go by a Neural Network Using Soft Segmentation. *Advances in Computer Games: Many Games, Many Challenges* (eds. H.J. van den Herik, H. Iida, and E.A. Heinz), pp. 97–108, Kluwer Academic Publishers, Boston, MA.

Evans, R. (2001). The Future of Game AI: A Personal View. *Game Developer Magazine*, Vol. 8, No. 8, pp. 46–49.

Evans, R. (2002). Varieties of Learning. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 567–578, Charles River Media, Inc., Hingham, MA.

Fairclough, C., Fagan, M., MacNamee, B., and Cunningham, P. (2001). Research Directions for AI in Computer Games. *12th Irish Conference on Artificial Intelligence & Cognitive Science (AICS 2001)* (ed. D. O'Donoghue), pp. 333–344.

Fleming, P.J. and Purhouse, R.C. (2001). Genetic Algorithms in Control Systems Engineering. Technical Report 789, University of Sheffield, Sheffield, UK.

Fogel, L.J. (1962). Autonomous Automata. *Industrial Research*, Vol. 4, pp. 14–19.

Forbus, K.D. and Laird, J. (2002). AI and the Entertainment Industry. *IEEE Intelligent Systems*, Vol. 17, No. 4, pp. 15–16.

Fu, D. and Houlette, R. (2004). Constructing a Decision Tree Based on Past Experiences. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 567–577, Charles River Media, Inc., Hingham, MA.

Funge, J.D. (2004). *Artificial Intelligence for Computer Games*. A K Peters, Ltd., Wellesley, MA.

Fyfe, C. (2004). Independent Component Analysis Against Camouflage. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass), pp. 259–262, University of Wolverhampton, Wolverhampton, UK.

Gold, J. (2004). *Object-oriented Game Development.* Addison-Wesley, Harrow, UK.

Goldberg, D.E. and Deb, K. (1991). A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. *Foundations of Genetic Algorithms* (ed. G.J.E. Rawlins), pp. 69–93, Morgan Kaufmann Publishers, San Francisco, CA.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley Publishing Company, Reading, MA.

Goldberg, D.E., Deb, K., and Korb, B. (1991). Don't Worry, Be Messy. *Fourth International Conference on Genetic Algorithms* (eds. R.K. Belew and L.B. Booker), pp. 24–30, Morgan Kaufmann Publishers, San Francisco, CA.

Graepel, T., Herbrich, R., and Gold, J. (2004). Learning to Fight. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass), pp. 193–200, University of Wolverhampton, Wolverhampton, UK.

Grefenstette, J. and Ramsey, C. (1992). An Approach to Anytime Learning. *Proceedings of the Ninth International Conference on Machine Learning* (eds. D.H. Sleeman and P. Edwards), pp. 189–195, Morgan Kaufmann, San Mateo, CA.

Halck, O.M. and Dahl, F.A. (1999). On Classification of Games and Evaluation of Players – with Some Sweeping Generalizations About the Literature. *Proceedings of the ICML-99 Workshop on Machine Learning in Game Playing* (eds. J. Fürnkranz and M. Kubat), J. Stefan Institute, Bled, Slovenia.

Hancock, P.J.B. (1992). Genetic Algorithms and Permutation Problems: a Comparison of Recombination Operators for Neural Structure Specification. *International Workshop on Combinations of Genetic Algorithms and Neural Networks* (eds. L.D. Whitley and J.D. Schaffer), pp. 108–122, IEEE Computer Society Press, Los Alamitos, CA.

Harmon, V. (2002). An Economic Approach to Goal-Directed Reasoning in an RTS. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 402–410, Charles River Media, Inc., Hingham, MA.

Hause, K. (1999). *What to Play Next: Gaming Forecast, 1999–2003.* Report W21056. International Data Corporation, Framingham, MA.

Heeter, C. (1992). Being There: The Subjective Experience of Presence. *Presence: Teleoperators and Virtual Environments*, Vol. 1, No. 2, pp. 262–271.

Herik, H.J. van den (1983). *Computerschaak, Schaakwereld en Kunstmatige Intelligentie.* Academic Service, 's Gravenhage, The Netherlands.

Herik, H.J. van den, Uiterwijk, J.W.H.M., and Rijswijck, J. van (2002). Games Solved: Now and in the Future. *Artificial Intelligence*, Vol. 134, pp. 277–311.

Herik, H.J. van den, Iida, H., and Heinz, E.A. (2003). *Advances in Computer Games: Many Games, Many Challenges*. Kluwer Academic Publishers, Boston, MA.

Hertz, J., Krogh, A., and Palmer, R.G. (1991). *An Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company, Reading, MA.

Holland, J.H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA.

Horn, J. (1997). Multicriterion Decision Making. *Handbook of Evolutionary Computation* (eds. T. Bäck, D. Fogel, and Z. Michalewicz), pp. F1.9:1–15. Oxford University Press, Oxford, UK.

Hsu, F-h. (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ.

Ierusalimschy, R., de Figueiredo, L.H., and Celes, W. (2003). *Lua 5.0 Reference Manual*. Technical Report MCC-14/03. PUC-Rio, Rio de Janeiro, Brazil.

Iida, H. and Yoshimura, J. (2003). *A Logistic Model of Game's Refinement*. CGRI Technical Report EG 2003–1. Department of Computer Science, Shizuoka University, Hamamatsu, Japan.

Iida, H., Handa, K-i, and Uiterwijk, J. (1995). Tutoring Strategies in Game-Tree Search. *ICCA Journal*, Vol. 18, No. 4, pp. 191–204.

Iida, H., Takeshita, N., and Yoshimura, J. (2002). A Metric for Entertainment of Boardgames: Its Implication for Evolution of Chess Variants. *Entertainment Computing: Technologies and Applications* (eds. R. Nakatsu and J. Hoshino), pp. 65–72, Kluwer Academic Publishers, Boston, MA.

Iida, H., Takahara, K., Nagashima, J., Kajihara, Y., and Hashimoto, T. (2004). An Application of Game-Refinement Theory to Mah Jong. *Entertainment Computing – ICEC 2004* (ed. M. Rauterberg), Lecture Notes in Computer Science 3166, pp. 333–338, Springer-Verlag, Berlin, Germany.

IJsselsteijn, W., de Kort, Y., Westerink, J., de Jager, M., and Bonants, R. (2004). Fun and Sports: Enhancing the Home Fitness Experience. *Entertainment Computing – ICEC 2004* (ed. M. Rauterberg), Lecture Notes in Computer Science 3166, pp. 46–56, Springer-Verlag, Berlin, Germany.

Jakobi, N. (1997). Evolutionary Robotics and the Radical Envelope of Noise Hypothesis. *Adaptive Behavior*, Vol. 6, No. 2, pp. 325–368.

Johnson, S. (2004). Adaptive AI: A Practical Example. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 639–647, Charles River Media, Inc., Hingham, MA.

Jones, J. and Goel, A. (2004). Hierarchical Judgement Composition: Revisiting the Structural Credit Assignment Problem. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 67–71, AAAI Press, Menlo Park, CA.

Karunanithi, N., Das, R., and Whitley, D. (1992). Genetic Cascade Learning for Neural Networks. *International Workshop on Combinations of Genetic Algorithms and Neural Networks* (eds. L.D. Whitley and J.D. Schaffer), pp. 134–145, IEEE Computer Society Press, Los Alamitos, CA.

Kent, T. (2004). Multi-Tiered AI Layers and Terrain Analyses for RTS games. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 447–455, Charles River Media, Inc., Hingham, MA.

Khoo, A. and Zubek, R. (2002). Applying Inexpensive AI Techniques to Computer Games. *IEEE Intelligent Systems*, Vol. 17, No. 4, pp. 2–7.

Kinnear, K.E. (1994). *Advances in Genetic Programming.* MIT Press, Cambridge, MA.

Kirby, N. (2004). Getting Around the Limits of Machine Learning. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 603–611, Charles River Media, Inc., Hingham, MA.

Knowles, B., Watamaniuk, P., Kristjanson, L., Soleski, K., Oster, T., McCabe, B., and Bishop, J. (2002). *Neverwinter Nights Manual.* BioWare Corp., Edmonton, Canada.

Kocsis, L. (2003). *Learning Search Decisions.* Ph.D. thesis, Universiteit Maastricht. Universitaire Pers Maastricht, Maastricht, The Netherlands.

Koller, D. and Pfeffer, A. (1997). Representations and Solutions for Game-Theoretic Problems. *Artificial Intelligence*, Vol. 94, No. 1, pp. 167–215.

Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA.

Laird, J.E. and Lent, M. van (2001). Human-Level's AI Killer Application: Interactive Computer Games. *Artificial Intelligence Magazine*, Vol. 22, No. 2, pp. 15–26.

Laird, J.E. (2000). Bridging the Gap Between Developers & Researchers. *Game Developers Magazine*, Vol. 8 (August).

Laird, J.E. (2001). It Knows What You're Going To Do: Adding Anticipation to a Quakebot. *Proceedings of the Fifth International Conference on Autonomous Agents* (eds. J.P. Müller, E. Andre, S. Sen, and C. Frasson), pp. 385–392, ACM Press, Montreal, Canada.

Laramée, F.D. (2002a). Genetic Algorithms: Evolving the Perfect Troll. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 629–639, Charles River Media, Inc., Hingham, MA.

Laramée, F.D. (2002b). Using N-Gram Statistical Models to Predict Player Behavior. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 596–601, Charles River Media, Inc., Hingham, MA.

Laurel, B. (1993). *Computers as Theatre*. Addison-Wesley Publishing Company, Reading, MA.

Le Diberder, A. and Le Diberder, F. (1993). *Qui A Peur des Jeux Vidéo?* Éditions La Découverte, Paris, France.

Le Hy, R., Arrigoni, A., Bessière, P., and Lebeltel, O. (2004). Teaching Bayesian Behaviours to Video Game Characters. *Robotics and Autonomous Systems*, Vol. 47, Nos. 2–3, pp. 177–185.

Lebling, P.D. (1980). Zork and the Future of Computerized Fantasy Simulations. *Byte*, Vol. 5, No. 12, pp. 172–182.

Ledwich, M. (2003). *Developing an Agent that Learns to Play Pacman*. B.Sc. thesis. University of Queensland, Queensland, Australia.

Lee, W-P., Hallam, J., and Lund, H.H. (1997). Applying Genetic Programming to Evolve Behavior Primitives and Arbitrators for Mobile Robots. *Proceedings of IEEE 4th International Conference on Evolutionary Computation*, pp. 495–499, IEEE Press.

Leen, G. and Fyfe, C. (2004). Agent Wars with Artificial Immune Systems. *Entertainment Computing – ICEC 2004* (ed. M. Rauterberg), Lecture Notes in Computer Science 3166, pp. 420–428, Springer-Verlag, Berlin, Germany.

Levy, S. (1984). *Hackers: Heroes of the Computer Revolution*. Penguin Putnam, New York, NY.

Lidén, L. (2002). Strategic and Tactical Reasoning with Waypoints. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 211–220, Charles River Media, Inc., Hingham, MA.

Lidén, L. (2004). Artificial Stupidity: The Art of Making Intentional Mistakes. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 41–48, Charles River Media, Inc., Hingham, MA.

Livingstone, D. and Charles, D. (2004). Intelligent Interfaces for Digital Games. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 6–10, AAAI Press, Menlo Park, CA.

Livingstone, D. and McGlinchey, S.J. (2004). What Believability Testing Can Tell Us. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass), pp. 273–277, University of Wolverhampton, Wolverhampton, UK.

Loe, C. and Crockett, F. (2002). *Neverwinter Nights Official Worldbuilder Guide*. Versus Books, Alameda, CA.

Louis, S.J. and Johnson, J. (1999). Robustness of Case-Initialized Genetic Algorithms. *Proceedings of the 12th International Florida Artificial Intelligence Research Symposium* (ed. AAAI Press California Edts), pp. 129–133.

Louis, S.J. and Li, G. (1997). Combining Robot Control Strategies Using Genetic Algorithms with Memory. *Evolutionary Programming VI* (eds. P.J. Angeline, R.G. Reynolds, J.R. McDonnell, and R. Eberhart), Lecture Notes in Computer Science 1213, pp. 431–441, Springer-Verlag, Berlin, Germany.

Louis, S.J. (2002). Learning from Experience: Case Injected Genetic Algorithm Design of Combinational Logic Circuits. *Adaptive Computing in Design and Manufacture V* (ed. I.C. Parmee), pp. 295–306, Springer-Verlag.

Madeira, C., Corruble, V., Ramalho, G., and Ratitch, B. (2004). Bootstrapping the Learning Process for the Semi-automated Design of Challenging Game AI. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 72–76, AAAI Press, Menlo Park, CA.

Man, K.F. and Tang, K.S. (1997). Genetic Algorithms for Control and Signal Processing. *23rd International Conference on Industrial Electronics, Control and Instrumentation*, Vol. 4, pp. 1541–1555.

Maniezzo, V. (1993). Searching Among Search Spaces: Hastening the Genetic Evolution of Feedforward Neural Networks. *Artificial Neural Nets and Genetic Algorithms* (eds. R.F. Albrecht, C.R. Reeves, and N.C. Steel), pp. 635–642, Springer-Verlag, Wien, Austria.

Manslow, J. (2002). Learning and Adaptation. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 557–566, Charles River Media, Inc., Hingham, MA.

Manslow, J. (2004). Using Reinforcement Learning to Solve AI Control Problems. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 591–601, Charles River Media, Inc., Hingham, MA.

Marthi, B., Latham, D., Russel, S., and Guestrin, C. (2004). Integrating Learning in Interactive Gaming Simulators. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 55–59, AAAI Press, Menlo Park, CA.

Matthews, K.B., Craw, S., Elder, S., Sibbald, A.S., and MacKenzie, I. (2000). Applying Genetic Algorithms to Multi-Objective Land Use Planning. *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 613–620, Morgan Kaufmann Publishers, San Francisco, CA.

McCulloch, W.S. and Pitts, W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, Vol. 5, pp. 115–133.

McGlinchey, S.J. (2003). Learning of AI Players from Game Observation Data. *Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)* (eds. Q. Mehdi, N. Gough, and S. Natkin), pp. 106–110, EUROSIS, Ghent, Belgium.

Mendel, G. (1866). Versuche über Pflanzen-Hybriden. *Verhandlungen des naturforschenden Vereines, Abhandlungen, Brünn*, Vol. 4, pp. 3–47.

Michalewicz, Z. and Fogel, D.B. (2000). *How To Solve It: Modern Heuristics*. Springer-Verlag, Berlin, Germany.

Michalewicz, Z. (1992). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, London, UK.

Minsky, M.L. and Papert, S.A. (1988). *Perceptrons: Introduction to Computational Geometry*. Expanded edition. MIT Press, Cambridge, MA.

Mitchell, T.M. (1997). *Machine Learning*. McGraw-Hill, Singapore, China, International edition.

Mommersteeg, F. (2002). Pattern Recognition with Sequential Prediction. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 586–595, Charles River Media, Inc., Hingham, MA.

Mondada, F., Franzi, E., and Jenne, P. (1993). Mobile Robot Miniaturisation: A Tool for Investigating in Control Algorithms. *Proceedings of the Third International Symposium on Experimental Robotics* (eds. T. Yoshikawa and F. Miyazaki), pp. 501–513, Springer-Verlag.

Montana, D. and Davis, L. (1989). Training Feedforward Neural Networks Using Genetic Algorithms. *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pp. 762–767, Morgan Kaufmann Publishers, San Francisco, CA.

Montfort, N. (2004). *Twisty Little Passages: An Approach to Interactive Fiction*. MIT Press, Cambridge, MA.

Nareyek, A. (2002). Intelligent Agents for Computer Games. *Computers and Games, Second International Conference, CG 2000* (eds. T.A. Marsland and I. Frank), Vol. 2063 of *Lecture Notes in Computer Science*, pp. 414–422, Springer-Verlag, Heidelberg, Germany.

Nareyek, A. (2004). AI in Computer Games. *ACM Queue*, Vol. 1, No. 10, pp. 58–65.

NWN Lexicon Group (2004). Neverwinter Nights Lexicon. www.nwnlexicon.com.

Ohlen, J., Kristjanson, L., Karpyshyn, D., and Muzyka, R. (2000). *Baldur's Gate II: Shadows of Amn Manual*. BioWare Corp., Edmonton, Canada.

Orkin, J. (2002). 12 Tips from the Trenches. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 29–35, Charles River Media, Inc., Hingham, MA.

Orkin, J. (2004a). Constraining Autonomous Character Behavior with Human Concepts. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 189–197, Charles River Media, Inc., Hingham, MA.

Orkin, J. (2004b). Simple Techniques for Coordinated Behavior. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 198–206, Charles River Media, Inc., Hingham, MA.

Pabst, T. (2000). 3D Benchmarking: Understanding Framerate Scores. graphics.tomshardware.com/graphic/20000704/.

Philips-Mahoney, D. (2002). Meeting of the Minds. *Computer Graphics World*, Vol. 25, No. 1, pp. 28–33.

Ponsen, M. and Spronck, P.H.M. (2004). Improving Adaptive Game AI with Evolutionary Learning. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass), pp. 389–396, University of Wolverhampton, Wolverhampton, UK.

Ponsen, M. (2004). *Improving Adaptive Game AI with Evolutionary Learning*. M.Sc. thesis. Delft University of Technology, Delft, The Netherlands.

Pyeatt, L.D. and Howe, A.E. (1998). Learning to Race: Experiments with a Simulated Race Car. *Proceedings of the Eleventh International Florida Artificial Intelligence Research Symposium Conference* (ed. D.J. Cook), pp. 357–361, AAAI Press, Sanibel Island, FL.

Rabin, S. (2004a). Filtered Randomness for AI Decisions and Game Logic. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 71–82, Charles River Media, Inc., Hingham, MA.

Rabin, S. (2004b). Promising Game AI Techniques. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 15–27, Charles River Media, Inc., Hingham, MA.

Ramsey, M. (2004). Designing a Multi-Tiered AI Framework. *AI Game Programming Wisdom 2* (ed. S. Rabin), pp. 457–466, Charles River Media, Inc., Hingham, MA.

Rijswijck, J. van (2003). Learning Goals in Sports Games. *Proceedings of the 2003 Game Developers Conference*, San Jose, CA. www.gdconf.com/archives/2003/Van_Ryswyck_Jack.doc.

Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, Vol. 65, pp. 386–408.

Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Pearson Education, Upper Saddle River, NJ, Second edition.

Sawyer, B. (2002). *Serious Games: Improving Public Policy through Game-based Learning and Simulation*. Foresight & Governance Project, Woodrow Wilson International Center for Scolars, Washington, DC. wwics.si.edu/topics/docs/ACF3F.pdf.

Schaeffer, J. and Herik, H.J. van den (2002). *Chips Challenging Champions: Games, Computers and Artificial Intelligence*. Elsevier, Amsterdam, The Netherlands.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag, New York, NY.

Schaeffer, J., Billings, D., Peña, L., and Szafron, D. (1999). Learning to Play Strong Poker. *Proceedings of the Sixteenth International Conference on Machine Learning*, J. Stefan Institute, Bled, Slovenia.

Schaeffer, J. (2001). A Gamut of Games. *Artificial Intelligence Magazine*, Vol. 22, No. 3, pp. 29–46.

Schaffer, J.D., Whitley, D., and Eschelman, L.J. (1992). Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art. *International Workshop on Combinations of Genetic Algorithms and Neural Networks* (eds. D. Whitley and J.D. Schaffer), pp. 1–37, IEEE Computer Society Press, Los Alamitos, CA.

Schapire, R. (2002). The Boosting Approach to Machine Learning: An Overview. *MSRI Workshop on Nonlinear Estimation and Classification* (eds. D.D. Denison, M.H. Hansen, C.C. Holmes, B. Mallick, and B. Yu), Lecture Notes in Statistics 171, pp. 149–172, Springer-Verlag, New York, NY.

Schwefel, H.-P. (1965). *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik*. M.Sc. thesis. Universität Berlin, Berlin, Germany.

Scott, B. (2002). The Illusion of Intelligence. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 16–20, Charles River Media, Inc., Hingham, MA.

Shah, I. (1968). *The Pleasantries of the Incredible Mulla Nasrudin*. John Carpenter Ltd, London, UK.

Slater, S. (2002). Enhancing the Immersive Experience. *Proceedings of the 3rd International Conference on Intelligent Games and Simulation* (eds. Q. Mehdi, N. Gough, and M. Cavazza), pp. 5–9, SCS Europe Bvba, Ghent, Belgium.

Snider, M. (2002). Where Movies End, Games Begin. *USA Today*, May 23, 2002.

Sprinkhuizen-Kuyper, I.G., Kortmann, R., and Postma, E.O. (2000a). Fitness Functions for Evolving Box-Pushing Behaviour. *Proceedings of the Twelfth Belgium-Netherlands Artificial Intelligence Conference* (eds. A. van den Bosch and H. Weigand), pp. 275–282.

Sprinkhuizen-Kuyper, I.G., Postma, E.O., and Kortmann, R. (2000b). Evolutionary Learning of a Robot Controller: Effect of Neural Network Topology. *Proceedings of the Tenth Belgian-Dutch Conference on Machine Learning* (ed. A. Feelders), pp. 55–60, Tilburg University.

Sprinkhuizen-Kuyper, I.G. (2001). *Artificial Evolution of Box-pushing Behaviour.* Report CS 01-02. Universiteit Maastricht, Faculty of General Sciences, IKAT/Department of Computer Science, Maastricht, The Netherlands.

Spronck, P.H.M. and Herik, H.J. van den (2003). Complex Games and Palm Computers. *Entertainment Computing: Technologies and Applications* (eds. R. Nakatsu and J. Hoshino), pp. 41–48, Kluwer Academic Publishers, Boston, MA.

Spronck, P.H.M. and Kerckhoffs, E.J.H. (1997). Using Genetic Algorithms to Design Neural reinforcement Controllers for Simulated Plants. *Proceedings of the 11th European Simulation Conference* (eds. A. Kaylan and A. Lehmann), pp. 292–299, SCS Europe Bvba, Erlangen, Germany.

Spronck, P.H.M. (1996). *Elegance: Genetic Algorithms in Neural Reinforcement Control.* M.Sc. thesis. Delft University of Technology, Delft, The Netherlands.

Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2001a). Infused Evolutionary Learning. *Proceedings of the Eleventh Belgian-Dutch Conference on Machine Learning* (eds. V. Hoste and G. de Pauw), pp. 61–68, University of Antwerp.

Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2001b). Island-based Evolutionary Learning. *Proceedings of the 13th Dutch-Belgian Artificial Intelligence Conference* (eds. B. Kröse, M. de Rijke, G. Schreiber, and M. van Someren), pp. 441–448, Universiteit van Amsterdam.

Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2002). Evolving Improved Opponent Intelligence. *GAME-ON 2002 3rd International Conference on Intelligent Games and Simulation* (eds. Q. Mehdi, N. Gough, and M. Cavazza), pp. 94–98, SCS Europe Bvba, Ghent, Belgium.

Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2003a). Improving Opponent Intelligence through Offline Evolutionary Learning. *International Journal of Intelligent Games and Simulation*, Vol. 2, No. 1, pp. 20–27.

Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2003b). Online Adaptation of Game Opponent AI in Simulation and in Practice. *Proceedings of*

the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003) (eds. Q. Mehdi and N. Gough), pp. 93–100, EUROSIS, Ghent, Belgium.

Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., Postma, E.O., and Kortmann, L.J. (2003c). Evolutionary Learning of a Box-Pushing Controller. *Computational Intelligence in Control* (eds. M. Mohammadian, R.A. Sarker, and X. Yao), pp. 104–121, Idea Group Publishing, Hershey, PA.

Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2004a). Difficulty Scaling of Game AI. *GAME-ON 2004 5th International Conference on Intelligent Games and Simulation* (eds. A. El Rhalibi and D. Van Welden), pp. 33–37, EUROSIS, Ghent, Belgium.

Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2004b). Enhancing the Performance of Dynamic Scripting in Computer Games. *Entertainment Computing – ICEC 2004* (ed. M. Rauterberg), Lecture Notes in Computer Science 3166, pp. 296–307, Springer-Verlag, Berlin, Germany.

Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2004c). Online Adaptation of Game Opponent AI with Dynamic Scripting. *International Journal of Intelligent Games and Simulation*, Vol. 3, No. 1, pp. 45–53.

Spronck, P.H.M., Sprinkhuizen-Kuyper, I.G., and Postma, E.O. (2005). DECA: The Doping-driven Evolutionary Control Algorithm. Submitted.

Spufford, F. (2003). Masters of their Universe. *The Guardian*, October 18, 2003.

Sterren, W. van der (2002). Squad Tactics: Team AI and Emergent Manoeuvres. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 233–246, Charles River Media, Inc., Hingham, MA.

Sutton, R.S. and Barto, A.G. (1998). *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA.

Tesauro, G. (1992). Practical Issues in Temporal Difference Learning. *Machine Learning*, Vol. 8, pp. 257–277.

Tesauro, G. (2002). Programming Backgammon Using Self-Teaching Neural Nets. *Artificial Intelligence*, Vol. 134, Nos. 1–2.

Thierens, D., Suykens, J., Vandewalle, J., and Moor, B. de (1993). Genetic Weight Optimization of a Feedforward Neural Network Controller. *Artificial Neural Nets and Genetic Algorithms* (eds. R.F. Albrecht, C.R. Reeves, and N.C. Steel), pp. 658–663, Springer-Verlag, Wien, Austria.

Tomlinson, S.L. (2003). Working at Thinking about Playing or A Year in the Life of a Games AI Programmer. *Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)* (eds. Q. Mehdi, N. Gough, and S. Natkin), pp. 5–12, EUROSIS, Ghent, Belgium.

Tozour, P. (2002a). Building an AI Diagnostic Toolset. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 39–45, Charles River Media, Inc., Hingham, MA.

Tozour, P. (2002b). The Evolution of Game AI. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 3–15, Charles River Media, Inc., Hingham, MA.

Tozour, P. (2002c). The Perils of AI Scripting. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 541–547, Charles River Media, Inc., Hingham, MA.

Tsang, E.P.K. (1993). *Foundations of Constraint Satisfaction*. Academic Press.

Turing, A.M. (1950). Computing Machinery and Intelligence. *Mind*, Vol. 59, No. 236, pp. 433–460.

Ulam, P., Goel, A., and Jones, J. (2004). Reflection in Action: Model-Based Self-Adaptation in Game Playing Agents. *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence* (eds. D. Fu, S. Henke, and J. Orkin), pp. 86–90, AAAI Press, Menlo Park, CA.

Veldhuizen, D.A. van and Lamont, G.B. (2000). Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art. *Evolutionary Computation*, Vol. 8, No. 2, pp. 125–147.

Wang, Q., Spronck, P., and Tracht, R. (2003). An Overview of Genetic Algorithms Applied to Control Engineering Problems. *Proceedings of the Second International Conference on Machine Learning and Cybernetics*, pp. 1651–1656.

Waveren, J.M.P. van and Rothkrantz, L.J.M. (2002). Artificial Player for Quake III Arena. *International Journal of Intelligent Games and Simulation*, Vol. 1, No. 1, pp. 25–32.

Werf, E.C.D. van der (2004). *AI Techniques for the Game of Go*. Ph.D. thesis, Universiteit Maastricht. Universitaire Pers Maastricht, Maastricht, The Netherlands.

Winands, H.H.M. (2004). *Informed Search in Complex Games*. Ph.D. thesis, Universiteit Maastricht. Universitaire Pers Maastricht, Maastricht, The Netherlands.

Woodcock, S. (1999). Game AI: The State of the Industry. *Game Developer Magazine*, Vol. 6, No. 8.

Woodcock, S. (2002). AI Roundtable Moderator's Report. www.gameai.com/cgdc02notes.html.

Wright, I. and Marshall, J. (2000). Egocentric AI Processing for Computer Entertainment: A Real-Time Process Manager for Games. *Proceedings of the 1st International Conference on Intelligent Games and Simulation* (eds. Q. Mehdi, N. Gough, and D. Al-Dabass), pp. 34–41, SCS Europe Bvba, Ghent, Belgium.

Yannakakis, G.N. and Hallam, J. (2004). Interactive Opponents Generate Interesting Games. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (eds. Q. Mehdi, N.E. Gough, S. Natkin, and D. Al-Dabass), pp. 240–247, University of Wolverhampton, Wolverhampton, UK.

Yao, X. (1995). Evolutionary Artificial Neural Networks. *Encyclopedia of Computer Science and Technology* (eds. A. Kent and J. Williams), Vol. 33, pp. 137–170, Marc Dekker, Inc., New York, NY.

Appendix A

CRPG Simulation Game AI

In Chapter 5, experiments with dynamic scripting in a simulated CRPG were discussed. This appendix describes implementation details of the CRPG simulation (A.1), the scripting language used to define game AI (A.2), the rulebases used to generate successful game AI for the dynamic team (A.3), and the tactics employed by the static team (A.4).

A.1 CRPG simulation

The CRPG simulation is modelled after the BALDUR'S GATE games. The implementations of agent attributes, combat, and magic are all to the specifications of BALDUR'S GATE II: SHADOWS OF AMN (Ohlen, Kristjanson, Karpyshyn, and Muzyka, 2000). The simulation entails an encounter between two teams of similar composition. Each team consists of four agents, namely two fifth-level 'fighters' and two fifth-level 'wizards'. The initial position of all agents in the CRPG simulation is illustrated in Figure 5.2. The front row of each team consists of the two fighters, and the back row of the two wizards. The combat area (the large square in which the agents are located) measures 1000×1000 units, which equals fifty by fifty feet. The initial distance between two fighters on opposite sides is 800 units.

The armament and weaponry of the teams is static, and each agent is allowed to carry two magic potions. In addition, the wizards are allowed to memorise seven magic spells. Potions and spells are implemented according to BALDUR'S GATE specifications (Ohlen *et al.*, 2000). Three different potions are available, namely of (i) Healing, (ii) Fire Resistance, and (iii) Free Action. Twenty-one magic spells are available, namely eight of the first level, eight of the second level, and five of the third level. The eight first-level spells are (i) Blindness, (ii) Charm Person, (iii) Chromatic Orb, (iv) Grease, (v) Larloch's Minor Drain, (vi) Magic Missile, (vii) Shield, and (viii) Shocking Grasp. The eight second-level spells are (i) Blur, (ii) Deafness, (iii) Luck, (iv) Melf's Acid Arrow, (v) Mirror Image, (vi) Ray of Enfeeblement, (vii) Stinking Cloud, and (viii) Strength. The five third-level spells are (i) Dispel Magic,

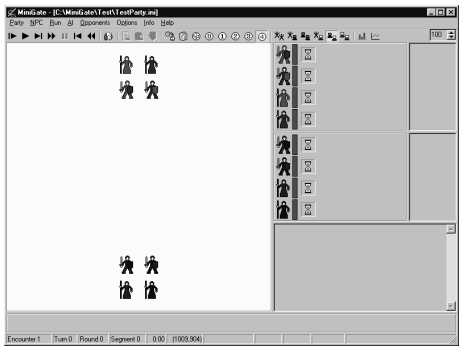

Figure A.1: The CRPG simulation.

(ii) Fireball, (iii) Flame Arrow, (iv) Hold Person, and (v) Monster Summoning I. A fifth-level wizard can memorise four first-level spells, two second-level spells, and one third-level spell.

A.2 Scripting Language

To implement game-AI scripts, the CRPG simulation employs a scripting language, which has been designed to be as powerful as the scripting language used for the BALDUR'S GATE games. It makes use of keywords and literals, which are listed in Table A.1. Besides the literals listed, names of potions and spells can also be used as literals. In the table, `self` refers to the agent whose script is executed, 'opponent agent' refers to a member of the team opposing `self`, and 'comrade agent' refers to a member of `self`'s team (including `self`). Game-AI scripts consist of a sequence of conditional statements, with an (optional) conditional part and an action part, structured as `if <conditional> then <action>`. When the game AI needs to select a new action, the statements in the script are checked in sequence. Of each statement, the conditional part is evaluated. If it evaluates to 'true' (or if it is absent), the corresponding action is checked. If the action obeys all relevant hard and soft constraints, it is selected and evaluation ends. Otherwise, the next statement in sequence is checked, until either an action is selected, or the script ends. The selected action is executed. If no action is selected, the default action `pass` is executed, though it is good practice to add actions to the end of the script that can always be executed.

The conditional part can check many different conditions, combined with the logical operators `and`, `or` and `not`. Conditions consist of either a logical method that returns a boolean, or a comparison between numerical expressions. The numerical expressions can use the numerical operators `+` (addition), `-` (substraction), `*` (multiplication), and `/` (division). Besides integers, the numerical expressions can use numerical methods.

Table A.1: Simulation scripting language: keywords and literals.

Actions	
`cast`	Called with a spell as parameter. Casts the spell.
`drink`	Called with a potion as parameter. Drinks the potion.
`meleeattack`	Called with an agent as parameter. Attacks the agent with the default melee weapon.
`movefrom`	Called with a location or an agent as parameter. Moves away in a direct line from the location, or from the agent.
`moveto`	Called with a location or an agent as parameter. Moves in a direct line towards the location, or towards the agent.
`pass`	Passes.
`rangedattack`	Called with an agent as parameter. Attacks the agent with the default ranged weapon.
Agents	
`closestenemy`	The opponent agent closest to `self`.
`closestfriend`	The comrade agent closest to `self`, excluding `self`.
`defaultenemy`	In the conditional statement, the most recently referred agent among the opponent agents.
`defaultfriend`	In the conditional statement, the most recently referred agent among the comrade agents.
`enemy`	Used with boolean methods; returns a random opponent agent for which the method returns true.
`friend`	Used with boolean methods; returns a random comrade agent for which the method returns true.
`furthestenemy`	The opponent agent furthest from `self`.
`furthestfriend`	The comrade agent furthest from `self`.
`randomenemy`	A random opponent agent.
`randomfriend`	A random comrade agent.
`self`	The agent whose script is executed.
`strongestenemy`	The opponent agent with the most health.
`strongestfriend`	The comrade agent with the most health.
`weakestenemy`	The opponent agent with the least health.
`weakestfriend`	The comrade agent with the least health.
Influences	
`badinfluence`	A detrimental influence.
`freezinginfluence`	A disabling influence.
`goodinfluence`	A beneficial influence.
Literals	
`"Acid"`	Influence. Caused by a 'Melf's Acid Arrow' spell.

continued on the next page

Table A.1: *continued from the previous page*

`"Animal"`	Agent type. Summoned monster.
`"Blinded"`	Influence. Caused by a 'Blindness' spell.
`"Blurred"`	Influence. Caused by a 'Blur' spell.
`"Charmed"`	Influence. Caused by a 'Charm Person' spell.
`"Deafened"`	Influence. Caused by a 'Deafness' spell.
`"Fighter"`	Agent type. Fighter class.
`"Fire Resistant"`	Influence. Caused by a potion of 'Fire Resistance'.
`"Freedom"`	Influence. Caused by a potion of 'Free Action'.
`"Held"`	Influence. Caused by a 'Hold Person' spell.
`"Lucky"`	Influence. Caused by a 'Luck' spell.
`"Mirrored"`	Influence. Caused by a 'Mirror Image' spell.
`"Nauseating Fumes"`	Cloud. Caused by a 'Stinking Cloud' spell.
`"Shielded"`	Influence. Caused by a 'Shield' spell.
`"Slippery Surface"`	Cloud. Caused by a 'Grease' spell.
`"Strengthened"`	Influence. Caused by a 'Strength' spell.
`"Stunned"`	Influence. Caused by a 'Chromatic Orb' spell or by a 'Nauseating Fumes' cloud.
`"Weakened"`	Influence. Caused by a 'Ray of Enfeeblement' spell.
`"Wizard"`	Agent type. Wizard class.
Locations	
`anywhere`	A random location anywhere in the combat area.
`backenemy`	Just behind the opponent agent furthest to the back.
`backfriend`	Just behind the comrade agent furthest to the back.
`centreall`	The mathematical centre of all agents.
`centreclouds`	The mathematical centre of all clouds in which the method-calling agent is located.
`centreenemy`	The mathematical centre of all opponent agents.
`centrefriend`	The mathematical centre of all comrade agents.
`frontenemy`	Just in front of the frontline opponent agent.
`frontfriend`	Just in front of the frontline comrade agent.
`randomenemyhalf`	A random location in the combat area at the side of the opponent team.
`randomfriendhalf`	A random location in the combat area at the side of the comrade team.
Methods	
`chancepercentage`	Called with a number as parameter. Returns 'true' with a chance equal to the parameter when it is interpreted as a percentage.

continued on the next page

Table A.1: *continued from the previous page*

`distance`	Called with one or two agents as parameter. With one agent as parameter, it returns the distance between that agent and the method-calling agent. With two agents as parameter, it returns the distance between the two agents.
`health`	The health of the method-calling agent as an integer.
`healthpercentage`	Called with a number as parameter. Returns the percentage that the current health of the method-calling agent is of its starting health.
`influence`	Called with an influence effect as parameter. Returns 'true' if the method-calling agent is under said effect.
`locatedin`	Called with a cloud effect. Returns 'true' if the method-calling agent is within the area covered by the cloud effect.
`maxhealth`	The initial health of the method-calling agent.
`random`	Called with a number as parameter. Returns a random integer less than the parameter.
`roundnumber`	The number of the current combat round.
`segmentnumber`	The number of the current combat-round segment.
`spellcount`	The number of spells the method-calling agent has memorised.
`stepsize`	The movement speed of the method-calling agent.
Potions	
`randompotion`	A random potion
Spells	
`randomareaeffect`	A random area-effect spell.
`randomcurse`	A random curse.
`randomdamaging`	A random damaging spell.
`randomdefensive`	A random defensive spell.
`randomoffensive`	A random curse or damaging spell.
`randomspell`	A random spell.
`strongareaeffect`	One of the highest-level area-effect spells.
`strongcurse`	One of the highest-level curses.
`strongdamaging`	One of the highest-level damaging spells.
`strongdefensive`	One of the highest-level defensive spells.
`strongoffensive`	One of the highest-level curses or damaging spells.
`weakareaeffect`	One of the lowest-level area-effect spells.
`weakcurse`	One of the lowest-level curses.
`weakdamaging`	One of the lowest-level damaging spells.
`weakdefensive`	One of the lowest-level defensive spells.
`weakoffensive`	One of the lowest-level curses or damaging spells.

Logical and numerical methods are called as <agent>.<method>(<parameters>). The agent whose script is executed can be referred to as self. If <agent> is self, the <agent>-part and the dot need not be included. If <method> does not have parameters, the part (<parameters>) can be ignored. Some methods are polymorphic, i.e., they have different implementations when used with different types of parameters.

Agents can be referred to using keywords. Except for defaultenemy, defaultfriend, and self, an agent keyword can be used with an agent-type literal, restricting the agent class to the value of the parameter.

As parameters, a method can take keywords and literals. 'Agent' parameters, 'influence' parameters, 'location' parameters, 'potion' parameters, and 'spell' parameters can be referred to using keywords. 'Influence' parameters, 'potion' parameters, and 'spell' parameters can also be referred to using literals. A numerical parameter is a numerical expression, which can contain numerical methods.

The action part of a conditional statement is called as a method, without specifying the <agent>, because it is always self that executes the action. Five actions are possible, namely (i) attacking (two varieties, namely with a melee weapon or with a ranged weapon), (ii) moving (two varieties, namely away from or towards), (iii) casting a spell, (iv) drinking a potion, and (v) passing.

A.3 Rulebases

In the simulated CRPG their are two classes of agents for which game AI can be defined, namely fighters and wizards. Each of these classes has its own rulebase for dynamic scripting to employ. The rulebase for fighters is presented in Subsection A.3.1, and the rulebase for wizards is presented in Subsection A.3.2.

A.3.1 Fighter Rulebase

This subsection presents the rulebase used by dynamic scripting for the fighter class in the simulated CRPG. The rulebase consists of twenty rules. In front of each rule are the rule number, and, between brackets, the priority of the rule. '[0]' is the lowest priority, while '[9]' is the highest priority.

```
1. [9] if roundnumber <= 1 then
           drink( "Potion of Fire Resistance" );
2. [9] if roundnumber <= 1 then
           drink( "Potion of Free Action" );
3. [5] if healthpercentage < 50 then
           drink( "Potion of Healing" );
4. [5] if healthpercentage < 25 then
           drink( "Potion of Healing" );
5. [5] if influence( "Slippery Surface" ) then
           drink( "Potion of Free Action" );
6. [3] movefrom( centreclouds );
```

```
 7. [3] if segmentnumber >= 1 then
            movefrom( centrefriend );
 8. [3] if locatedin( "Nauseating Fumes" ) then
            drink( "Potion of Free Action" );
 9. [1] meleeattack( closestenemy( "Wizard" ) );
10. [1] meleeattack( closestenemy( "Fighter" ) );
11. [1] if distance( weakestenemy ) > 300 then
            rangedattack( defaultenemy );
12. [1] if distance( weakestenemy( "Wizard" ) ) > 300 then
            rangedattack( defaultenemy );
13. [1] if not influence( "Slippery Surface" ) then
            meleeattack( closestenemy );
14. [1] if distance( closestenemy ) > 300 then
            rangedattack( randomenemy );
15. [1] if distance( closestenemy ) > 300 then
            rangedattack( weakestenemy );
16. [1] if distance( closestenemy ) < 200 then
            meleeattack( defaultenemy );
17. [1] drink( randompotion );
18. [0] meleeattack( weakestenemy );
19. [0] rangedattack( weakestenemy );
20. [0] meleeattack( closestenemy );
```

Rule 1 and 2 force the agent to perform a specific action in the very first round, but not later. These rules have the highest priority, because they are only useful when at the very beginning of the script.

Rule 6 states that the agent should move away from the centre of a cloud. The location `centreclouds` only returns a valid value for the action `movefrom` if the agent is actually located in a cloud. All clouds in the CRPG simulation have a detrimental effect, and rule 6 helps agents to avoid them.

Rule 7 checks a segment number. A combat round consists of ten segments. In the first segment of a combat round an agent choses an action, which is executed in one of the later segments (it depends on the action when that will be exactly). After an action is executed, an agent has to wait until the next round to choose a new action. However, the agent still has the ability to move. Rule 7 gives an agent extra move actions after the agent's main action for the combat round has been executed.

A fighter game-AI script consists of five rules extracted from the rulebase, to which at the end the rule `meleeattack( closestenemy )` is attached.

A.3.2 Wizard Rulebase

This subsection presents the rulebase used by dynamic scripting for the wizard class in the simulated CRPG. The rulebase consists of fifty rules. In front of each rule are the rule number, and, between brackets, the priority of the rule. '[0]' is the lowest priority, while '[9]' is the highest priority.

```
 1. [9] if influence( "Acid" ) then
            rangedattack( closestenemy( "Wizard" ) );
 2. [9] if roundnumber <= 1 then
            drink( "Potion of Fire Resistance" );
 3. [9] if roundnumber <= 1 then
            drink( "Potion of Free Action" );
 4. [9] if roundnumber <= 1 then
            cast( "Monster Summoning I", centreenemy );
 5. [9] if roundnumber <= 1 then
            cast( "Hold Person", randomenemy );
 6. [9] if roundnumber <= 1 then
            cast( "Fireball", centreenemy );
 7. [9] if roundnumber <= 1 then
            cast( "Mirror Image" );
 8. [7] if roundnumber <= 1 then
            cast( randomdefensive );
 9. [5] if locatedin( "Nauseating Fumes" ) then
            drink( "Potion of Free Action" );
10. [5] if enemy.influence( "Charmed" ) then
            cast( "Charm Person", defaultenemy );
11. [3] if healthpercentage < 50 then
            drink( "Potion of Healing" );
12. [3] movefrom( centreclouds );
13. [3] if segmentnumber >= 1 then
            movefrom( centrefriend );
14. [3] if segmentnumber >= 1 then
            movefrom( closestenemy );
15. [3] if friend.influence( badinfluence ) and
            not defaultfriend.influence( goodinfluence ) then
            cast( "Dispel Magic", defaultfriend );
16. [2] cast( "Fireball", furthestenemy );
17. [2] cast( "Charm Person", randomenemy( "Fighter" ) );
18. [2] cast( "Charm Person", randomenemy( "Wizard" ) );
19. [2] cast( "Deafness", randomenemy( "Wizard" ) );
20. [2] cast( "Monster Summoning I", randomenemyhalf );
21. [2] cast( "Ray of Enfeeblement", randomenemy( "Fighter" ) );
22. [2] if friend.influence( "Weakened" ) then
            cast( "Strength", defaultfriend );
23. [2] if friend( "Wizard" ).influence( "Deafened" ) then
            cast( "Dispel Magic", defaultfriend );
24. [2] cast( "Mirror Image" );
25. [2] cast( "Blindness", randomenemy( "Fighter" ) );
26. [2] cast( "Blur" );
27. [2] cast( "Shield" );
28. [2] cast( "Luck", randomfriend );
```

```
29. [2] cast( "Chromatic Orb", randomenemy );
30. [2] if roundnumber <= 1 then
            cast( "Stinking Cloud", centreenemy );
31. [2] cast( "Stinking Cloud", randomenemy( "Wizard" ) );
32. [2] cast( "Stinking Cloud", randomenemy( "Fighter" ) );
33. [2] cast( "Hold Person", closestenemy );
34. [2] cast( "Flame Arrow", randomenemy );
35. [2] if (health < maxhealth - 4) and (weakestenemy.health >= 4) then
            cast( "Larloch's Minor Drain", defaultenemy );
36. [2] cast( "Grease", randomenemy( "Fighter" ) );
37. [2] cast( "Magic Missile", weakestenemy( "Wizard" ) );
38. [2] cast( "Magic Missile", weakestenemy );
39. [2] cast( "Melf's Acid Arrow", randomenemy( "Wizard" ) );
40. [2] cast( "Shocking Grasp", closestenemy );
41. [2] cast( "Blur" );
42. [1] cast( randomoffensive, randomenemy );
43. [1] cast( randomblessing, randomfriend );
44. [1] cast( randomcurse, randomenemy );
45. [1] cast( randomdefensive );
46. [1] cast( randomareaeffect, randomenemy );
47. [1] drink( randompotion );
48. [0] rangedattack( weakestenemy( "Wizard" ) );
49. [0] rangedattack( weakestenemy );
50. [0] if distance( closestenemy ) < 100 then
            meleeattack( defaultenemy );
```

Rule 1 forces the agent to use a ranged weapon to attack, when under the influence of acid. Acid damage causes any spell the wizard has selected to fail. Therefore, whilst under the influence of acid, spell-casting is not useful. Rule 1 takes this into account by forcing the wizard to use ranged attacks until the acid has dissolved.

Rule 6 forces the agent to cast a 'Fireball' spell the very first round. A 'Fireball' is an area-effect spell, which seriously damages anyone in its range of effect. It is most useful against a group of opponents that are standing close together, while comrades are still a good distance away. This is the situation at the start of combat.

Rule 10 checks whether there is an opponent that is charmed. An opponent that is charmed, is actually a friend under the influence of a 'Charm Person' spell, who is now fighting for the opposing team. A second 'Charm Person' spell cast at the opponent will remove the effect of the first spell, turning the erstwhile opponent friendly again.

Rule 15 checks whether a comrade is under any detrimental spell effect, while not being under any beneficial spell effect. If so, the wizard attempts to remove several detrimental spell effects with the 'Dispel Magic' spell. Since 'Dispel Magic' makes no difference between detrimental and beneficial spell effects, 'Dispel Magic' is best applied at a comrade that is only affected by detrimental effects. The rule takes this into account.

Rule 19 makes the agent cast 'Deafness' at an opponent wizard. While 'Deafness' can be cast at fighters, it only affects wizards detrimentally.

Rule 21 makes the agent cast 'Ray of Enfeeblement' at an opponent fighter. 'Ray of Enfeeblement' saps the strength of an opponent. While 'Ray of Enfeeblement' can be cast at wizards, wizards do not have high strength to begin with. Therefore, the spell is most useful against fighters.

Rule 23 makes the agent cast 'Dispel Magic' to a comrade wizard that suffers from the 'Deafness' spell. Within the CRPG simulation, 'Dispel Magic' is the only remedy against being deafened.

Rule 30 is actually a mistake; it should have priority 9, but it has priority 2. When this rule is selected for a script, its chance to be activated is remote.

A wizard game-AI script consists of ten rules extracted from the rulebase, to which at the end the rules `cast( strongoffensive, closestenemy )` and `rangedattack( closestenemy )` are attached.

A.4 Static Tactics

Chapter 5 refers to five different basic tactics used by the static team. The tactics consist of a game-AI script for each of the members of the static team. The team consists of two fighters and two wizards. For all tactics, the two fighters use the same script. The following five subsections present the scripts used for each of the five static tactics, namely the 'offensive' tactic (A.4.1), the 'disabling' tactic (A.4.2), the 'cursing' tactic (A.4.3), the 'defensive' tactic (A.4.4), and the 'novice' tactic (A.4.5).

A.4.1 The Offensive Tactic

For the 'offensive' tactic, the two fighters use the following script:

```
if healthpercentage < 50 then
    drink( "Potion of Healing" );
meleeattack( closestenemy );
```

With the 'offensive' tactic, the two fighters will use their melee weapon to attack opponents. In general, fighters are much more effective when using melee attacks than when using ranged attacks. The fighters will attempt to heal when they are damaged too much.

The two wizards both use the following script:

```
if healthpercentage < 50 then
    drink( "Potion of Healing" );
cast( "Fireball", centreenemy );
cast( "Melf's Acid Arrow", closestenemy( "Wizard" ) );
cast( "Melf's Acid Arrow", closestenemy );
cast( "Magic Missile", weakestenemy );
rangedattack( closestenemy );
```

With the 'offensive' tactic, the very first round of an encounter, both wizards will throw a 'fireball' at the centre the opponent team. The effect is that usually the two wizards of the opposing team will be killed outright, unless they immediately start moving or take protective measures. In the following rounds, the two wizards will first attempt to kill opponents with damaging magic spells, starting any remaining opponent wizard. When the wizards are out of spells, they will use ranged attacks.

A.4.2 The Disabling Tactic

For the 'disabling' tactic, the two fighters use the following script:

```
if roundnumber <= 1 then
    drink( "Potion of Free Action" );
if healthpercentage < 50 then
    drink( "Potion of Healing" );
meleeattack( closestenemy );
```

With the 'disabling' tactic, the two fighters will first drink a potion of free action, ensuring that they will be unaffected by the area-effect spells used by the wizards in the team. The remainder of the script is equal to the offensive tactic script.

The first wizard uses the following script:

```
if healthpercentage < 50 then
    drink( "Potion of Healing" );
drink( "Potion of Free Action" );
if not closestenemy( "Fighter" ).influence( freezinginfluence ) then
    cast( "Stinking Cloud", defaultenemy );
cast( "Chromatic Orb", closestenemy( "Fighter" ) );
cast( "Hold Person", randomenemy );
cast( "Stinking Cloud", randomenemy );
cast( "Chromatic Orb", randomenemy );
rangedattack( closestenemy );
```

The second wizard uses the same script, except that in lines 4 and 6, the references to "Fighter" are replaced by "Wizard". With the 'disabling' tactic, the two wizards will first drink a potion of free action, ensuring that they will be unaffected by the area-effect spells they use.[1] After that they use all kinds of spells that disable their opponents, such as freezing them in place, or making them nauseous. When the wizards are out of spells, they will use ranged attacks.

[1] As Chapter 5 showed, the 'disabling' tactic is rather weak. The main reason for its weakness is that all four static-team members drink a potion in the first combat round. Since they do not move from their starting position, they are rather susceptible to their opponents attacking them with damaging area-effect magic, similar to the 'offensive' tactic.

A.4.3 The Cursing Tactic

For the 'cursing' tactic, the two fighters use the same script as with the 'offensive' tactic. The first wizard uses the following script:

```
if healthpercentage < 50 then
    drink( "Potion of Healing" );
cast( "Hold Person", closestenemy( "Fighter" ) );
cast( "Deafness", closestenemy( "Wizard" ) );
cast( "Charm Person", closestenemy( "Wizard" ) );
cast( "Ray of Enfeeblement", closestenemy( "Fighter" ) );
cast( "Blindness", closestenemy( "Fighter" ) );
if not furthestenemy( "Fighter" ).influence( freezinginfluence ) then
    cast( "Chromatic Orb", defaultenemy );
if not furthestenemy( "Wizard" ).influence( freezinginfluence ) then
    cast( "Chromatic Orb", defaultenemy );
cast( "Chromatic Orb", randomenemy );
rangedattack( closestenemy );
```

The second wizard uses the following script:

```
if healthpercentage < 50 then
    drink( "Potion of Healing" );
cast( "Monster Summoning I", centreenemy );
cast( "Deafness", closestenemy( "Wizard" ) );
cast( "Charm Person", closestenemy( "Fighter" ) );
cast( "Ray of Enfeeblement", closestenemy( "Fighter" ) );
cast( "Blindness", closestenemy( "Fighter" ) );
if not closestenemy( "Wizard" ).influence( freezinginfluence ) then
    cast( "Chromatic Orb", defaultenemy );
if not closestenemy( "Fighter" ).influence( freezinginfluence ) then
    cast( "Chromatic Orb", defaultenemy );
cast( "Chromatic Orb", randomenemy );
rangedattack( closestenemy );
```

The 'cursing' tactic aims at the wizards hampering their opponents in several different ways, while the fighters attack them up-close. While the two wizards mostly use the same spells, they attempt to chose different targets for their spells. The 'cursing' tactic relies heavily on chance. Especially the use of charming spells is risky: they have a 50 per cent chance to fail. However, if they succeed, they can be decisive in determining the outcome of the fight. The 'cursing' tactic is quite strong if chance is in favour of the static team, but it is mediocre otherwise. As a result, the 'cursing' tactic is most susceptible to the occurrence of extreme outliers.

A.4.4 The Defensive Tactic

For the 'defensive' tactic, the two fighters use the following script:

```
if roundnumber <= 1 then
    drink( "Potion of Fire Resistance" );
if healthpercentage < 50 then
    drink( "Potion of Healing" );
meleeattack( closestenemy );
```

With the 'defensive' tactic, the two fighters will first drink a potion of fire resistance, ensuring that fire-damaging spells, which are the most common damaging spells at this level, are less effective when used against them. The remainder of the script is equal to the offensive tactic script.

The first wizard uses the following script:

```
if healthpercentage < 50 then
    drink( "Potion of Healing" );
cast( "Mirror Image" );
cast( "Monster Summoning I", centreenemy );
cast( "Shield" );
cast( "Larloch's Minor Drain", closestenemy );
rangedattack( closestenemy );
```

The second wizard uses the same script, except that line 5 is replaced by "`cast( "Fireball", closestenemy( "Fighter" ) );`". The 'defensive' tactic aims at the static team's wizard using mainly defensive spells. Especially the 'Mirror Image' spell is, in the BALDUR'S GATE implementation,[2] quite effective in keeping the wizards from suffering any damage.

A.4.5 The Novice Tactic

For the 'novice' tactic, the two fighters use the same script as with the 'offensive' tactic. The first wizard uses the following script:

```
if healthpercentage < 50 then
    drink( "Potion of Healing" );
cast( "Hold Person", closestenemy( "Fighter" ) );
cast( "Mirror Image" );
if not closestenemy( "Fighter" ).influence( freezinginfluence ) then
    cast( "Stinking Cloud", defaultenemy );
cast( "Magic Missile", closestenemy( "Wizard" ) );
cast( randomoffensive, randomenemy );
cast( "Chromatic Orb", randomenemy );
rangedattack( closestenemy );
```

The second wizard uses the following script:

[2] The BALDUR'S GATE implementation of the 'Mirror Image' spell is actually quite different from official specification (Cook *et al.*, 2000); so much, in fact, that the BALDUR'S GATE implementation may be considered a programming bug, for the spell is much too powerful for the level at which it is available in the game.

```
if healthpercentage < 50 then
    drink( "Potion of Healing" );
cast( "Mirror Image" );
cast( "Fireball", closestenemy( "Wizard" ) );
cast( randomoffensive, randomenemy );
rangedattack( closestenemy );
```

The 'novice' tactic aims at imitating a tactic that a novice player might use. A novice player will probably have discovered the power of the 'Mirror Image' spell and the 'Fireball' spell, but other than that will not know which spells are effective and which are not. In the tactic, this is implemented as the wizards using mostly random spells.

Appendix B

Neverwinter Nights Game AI

In Chapter 5, experiments with dynamic scripting in the game NEVERWINTER NIGHTS were discussed. This appendix describes NEVERWINTER NIGHTS and the module implemented for the experiments (B.1), the static game AI implemented by the game developers (B.2), and the rulebases used to generate successful game AI for the dynamic team (B.3).

B.1 Neverwinter Nights Module

NEVERWINTER NIGHTS is a CRPG, developed by BioWare Corp (located in Edmonton, Canada), released in 2002. One of the major gimmicks of the game is the availability of an extensive toolset, called 'Aurora', that can be used to develop completely new game modules based on the NEVERWINTER NIGHTS game engine. Aurora scales fairly well from novice users without programming experience, who can easily fit together existing game elements, to experienced programmers, who can rebuild the inner workings of the game from scratch. BioWare proved the power of the toolset, by commercially releasing two new NEVERWINTER NIGHTS modules in 2003, which were developed by a third party.

The NEVERWINTER NIGHTS module developed to perform the experiments discussed in Chapter 5 entails an encounter between two teams of similar composition. Each team consists of four agents, namely a fighter, a priest, a rogue, and a wizard, all of the eighth experience level. The initial position of all agents in the CRPG simulation is illustrated in Figure B.1. The front row of each team consists of the fighter and the priest, and the back row of the wizard and the rogue. The combat area (the arena in which the agents are located) has a diameter of one-and-one-half NEVERWINTER NIGHTS cells, or fifty feet.

The armament, weaponry, spell selection and inventory of the teams is static. Each fighter carries a potion of 'Cure Serious Wounds' and a potion of 'Speed'. Each wizard carries a potion of 'Cure Light Wounds' and a potion of 'Speed'. Each priest carries a potion of 'Cure Moderate Wounds', a potion of 'Owl's Wisdom', and a

Figure B.1: The NEVERWINTER NIGHTS module.

potion of 'Bless'. Each rogue carries a potion of 'Cure Moderate Wounds', a potion of 'Speed', and a potion of 'Invisibility'. Wizards have access to the following spells (one copy of each spell, unless indicated otherwise): 'Daze' (two copies), 'Ray of Frost', 'Resistance', 'Burning Hands', 'Magic Missile' (two copies), 'Negative Energy Ray', 'Melf's Acid Arrow' (two copies), 'Summon Creature II', 'Fireball', 'Flame Arrow', 'Negative Energy Burst', 'Evard's Black Tentacles', and 'Minor Globe of Invulnerability'. Priests have access to the following spells (one copy of each spell, unless indicated otherwise): 'Cure Minor Wounds', 'Light' (two copies), 'Resistance', 'Virtue' (two copies), 'Cure Light Wounds', 'Doom', 'Sanctuary', 'Summon Creature I', 'Aid', 'Silence', 'Sound Burst', 'Animate Dead', 'Cure Serious Wounds', 'Prayer', 'Cure Critical Wounds', 'Divine Power'. A detailed description of NEVERWINTER NIGHTS is given by Knowles *et al.* (2002).

I chose not to include a 'sorcerer' in the teams. The reason is that sorcerers are not limited to the spells they memorise, but can use any of the spells of the levels they have access to. Therefore, a sorcerer can always execute the first rule in a script that casts a spell, and will continue casting the same spell over and over again until all casting power is gone. Therefore, for a sorcerer, scripting is not ideal.

As an alternative, a sorcerer could be controlled by the rulebase as a whole, where for each action a rule is selected at random from the rulebase, with a probability corresponding to the rules' weights. This system has actually been implemented in the NEVERWINTER NIGHTS module as an alternative to the scripting system, but no experiments have been performed with it yet.

B.2 Static Game AI

The NEVERWINTER NIGHTS game AI is implemented in the NEVERWINTER NIGHTS scripting language called 'NWScript'. NWScript is derived from C++. Although it lacks many of the powerful features of C++,[1] it is a fairly powerful language that allows the implementation of advanced concepts. NWScript is documented by Loe and Crockett (2002) and by the NWN Lexicon Group (2004).

The NEVERWINTER NIGHTS game AI is implemented in NWScript. This section discusses the three different variations of the NEVERWINTER NIGHTS game AI used in this thesis, namely (i) the game AI of NEVERWINTER NIGHTS version 1.29 (B.2.1), (ii) the game AI of NEVERWINTER NIGHTS version 1.61 (B.2.2), and (iii) the cursed version of the game AI of NEVERWINTER NIGHTS version 1.61 (B.2.2),

B.2.1 Game AI 1.29

The game AI included in NEVERWINTER NIGHTS version 1.29 consists of a straightforward script, titled `DetermineCombatRound()` (found in the file `nw_i0_generic`). executed for all agents in the game. Basically, each line of the script consists of a check whether the class of the agent is allowed to execute that line (e.g., a line concerning magic will only be executed for spell casters), followed by a 'talent'. A 'talent' is a call to a function that may perform an action of a certain type. If the talent indeed generates an action, it returns the value 'true' and the script ends. If not, it returns the value 'false' and the next line in the script is executed. For instance, the following is a short code snippet from the game AI script:

```
if (nClass == CLASS_TYPE_BARD)
{
    if (TalentHeal())
        return;
    if (TalentBardSong())
        return;
}
```

This code tests whether the class of the agent that executes the script is 'bard'. If so, then the function `TalentHeal()` is called. This function checks whether the agent has healing capabilities, and whether it is useful at this point to perform a healing action. If no healing action is generated, the function `TalentBardSong()` is called,

[1] For instance, other than 'string', 'integer' and 'float' there are no variable types, and it is not possible to create new classes.

which checks whether it is useful at this point for the agent to perform a singing action.

The game AI uses random numbers to provide variety. For instance, at the start of the script a random number decides whether the agent will perform an offensive (with 75 per cent probability) or a defensive action (with 25 per cent probability).

The game AI of NEVERWINTER NIGHTS version 1.29 is not so strong. For instance, when computer-controlled agents are distanced further from their enemies than they can cover in one combat round, they will use ranged weapons. They will stick to using ranged weapons, even if their enemy closes in. Since usually agents do more damage with melee weapons than with ranged weapons, an effective way to deal with agents using ranged weapons is to run towards them and attack with melee weapons. This is actually one of the tactics discovered by dynamic scripting against game AI 1.29.

B.2.2 Game AI 1.61

The NEVERWINTER NIGHTS game AI was completely rewritten about a year after the first release of the game. The game AI for version 1.61 is significantly more effective than the game AI for version 1.29.

Game AI 1.61 starts by assigning integer values to three variables, named `nOffense`, `nCompassion`, and `nMagic`. These variables represent a percentage probability to use an offensive attack, to help companions, and to use magic, respectively. A fourth variable, named `nCrazy`, is a modifier that decides how big the variety in decisions is. The variables get typical values for the class and attributes of the agent for which the game AI is executed. Then, the values of the variables are used to decide which part of the script is executed. For instance, the following is a short code snippet from the game AI script:

```
if ((nOffense <= 50) && (nMagic > 50) && (nCompassion > 50))
{
    if (TalentHeal())
        return;
    if (TalentCureCondition())
        return;
    if (TalentUseProtectionOthers())
        return;
    if (TalentEnhanceOthers())
        return;
}
```

This code tests whether the agent is not offensive, has access to magic, and feels compassionate. If so, it attempts to select a 'talent' that supports its companions. It first attempts healing, then curing (e.g., removing poison), then protective magic, and finally general enhancements of others.

The game AI provides variety by using random values for the four variables, ensuring that the values which the variables receive are in accordance with the class

and attributes of the agent that executes the script. The talents themselves have been updated to remove some randomness, and to make them more effective.

The game AI of NEVERWINTER NIGHTS version 1.61 is considerably stronger than the game AI of NEVERWINTER NIGHTS version 1.29. For instance, fighter agents that are able to use strong melee attacks, will often attack with melee weapons, even if they start out far from their enemies. They are also more limited in their ability to choose less effective actions. For instance, while in NEVERWINTER NIGHTS version 1.29 they often wasted time by drinking useless potions, in NEVERWINTER NIGHTS version 1.61 fighters will never drink potions except to heal.

Interestingly, the reduced amount of randomness allows dynamic scripting to design tactics that are able to easily defeat game AI 1.61. For instance, a dynamic fighter agent will quickly learn to drink a potion of 'Speed' at the start of a fight, allowing it more effective melee attacks than a static fighter agent that refuses to drink any potion.

B.2.3 Cursed Game AI

Cursed game AI is actually equal to game AI 1.61. However, there is a difference in the way the combat is handled. With cursed game AI, after every twelve fights, three 'cursed' fights are executed. At the start of a cursed fight, the average fitness for both teams over the last ten fights is calculated. If the dynamic team has a higher average fitness than the static team, the static team gets cursed, otherwise the dynamic team gets cursed. The cursing of a team consists of disabling the members of the team for the first 60 seconds of a fight. Furthermore, if the static team is cursed, the dynamic team selects rules from the rulebase using all equal weights.

Consequently, when the dynamic team is winning (i.e., has a higher average fitness), during the cursed fights it will be at a great disadvantage to the static team. Therefore, it is likely that a dynamic team that employs a successful rulebase AI will lose a cursed fight despite using good AI. Contrariwise, when the dynamic team is losing (i.e., has a lower average fitness), during the cursed fights it will be at a great advantage to the static team, and thus will probably win despite using random AI.

In summary, for 20 per cent of the fights, cursed game AI attempts to fool dynamic scripting into rating good AI as being inferior, and rating random AI as being good.

B.3 Rulebase

Dynamic scripting as implemented in NEVERWINTER NIGHTS uses one central rulebase for all classes. For each rule in the rulebase an indication is given for which classes the rule is meant. At the start of a test (i.e., a series of fights), a separate rulebase is created for each class by extracting those rules from the central rulebase corresponding to the class.

The central rulebase is listed below. In front of each rule are the rule number, and, between brackets, the priority of the rule. '[0]' is the lowest priority, while '[4]' is the highest priority. Instead of code, a description of each rule is given, followed by the classes for which the rule is applicable. 'F' indicates the fighter class, 'P' indicates the priest class, 'R' indicates the rogue class, and 'W' indicates the wizard class. The implementation of the rules is always by calling a 'talent' function, in many cases the same 'talent' functions the standard game AI uses.

1. [4] Heal self when health < 25% (F,P,R,W)
2. [4] If not yet in combat, buff self (F,R)
3. [4] Cast 'Immunity to Death Magic' (P)
4. [4] Cast 'Freedom' (P)
5. [4] Cast 'Regenerate' (P)
6. [4] Cast 'Mass Haste' or 'Haste' (P,W)
7. [4] Cast 'Time Stop' (W)
8. [4] Heal self when health < 50% (F,P,R,W)
9. [4] Empty rule (F,P,R,W)
10. [3] Cast highest magic-absorption spell (W)
11. [3] Cast highest summoning spell at nearest enemy (P,W)
12. [3] Cast highest summoning spell at nearest enemy spellcaster (P,W)
13. [3] Cast highest area-effect damaging spell at nearest enemy (P,W)
14. [3] Cast highest area-effect damaging spell at nearest enemy spellcaster (P,W)
15. [3] Cast highest damaging-cloud spell at nearest enemy (P,W)
16. [3] Cast highest damaging-cloud spell at nearest enemy spellcaster (P,W)
17. [3] Cast highest cursing-cloud spell at nearest enemy (P,W)
18. [3] Cast highest cursing-cloud spell at nearest enemy spellcaster (P,W)
19. [3] Cast highest area-effect cursing spell at nearest enemy (P,W)
20. [3] Cast highest area-effect cursing spell at nearest enemy spellcaster (P,W)
21. [3] Cast highest cone spell at nearest enemy (P,W)
22. [3] Cast highest cone spell at nearest enemy spellcaster (P,W)
23. [3] Cast highest damaging spell at nearest enemy (P,W)
24. [3] Cast highest damaging spell at nearest enemy spellcaster (P,W)
25. [3] Cast highest cursing spell at nearest enemy (P,W)
26. [3] Cast highest cursing spell at nearest enemy spellcaster (P,W)
27. [3] Cast highest anti-invisibility spell (P,W)
28. [3] Cast highest anti-mind-affecting spell (P,W)
29. [3] Cast highest damage-absorption spell (P,W)
30. [3] Cast highest breach spell at nearest enemy (P,W)
31. [3] Cast highest breach spell at nearest enemy spellcaster (P,W)

```
32. [3] Melee-attack nearest enemy (F,R)
33. [3] Melee-attack nearest enemy spellcaster (F,R)
34. [3] Ranged-attack nearest enemy (F,R)
35. [3] Ranged-attack nearest enemy spellcaster (F,R)
36. [3] Melee-attack nearest enemy fighter or rogue (F,R)
37. [3] Ranged-attack nearest enemy fighter or rogue (F,R)
38. [3] Empty rule (F,P,R,W)
39. [2] Heal a companion (P)
40. [2] Heal self (F,P,R,W)
41. [2] Use advanced protective magic on self (P,W)
42. [2] Use protective magic on self (P,W)
43. [2] Use protective magic on companions (P,W)
44. [2] Buff self (F,P,R,W)
45. [2] Buff companions (P,W)
46. [2] Respond to a melee-attacker against self, preferably a
        spellcaster (P,W)
47. [2] Respond to a ranged-attacker against self, preferably a
        spellcaster (P,W)
48. [2] Use offensive magic at an enemy that attacks from a distance,
        preferably a spellcaster (P,W)
49. [2] Use summoning magic (P,W)
50. [2] Use offensive magic against the nearest spellcaster (P,W)
51. [2] Melee-attack nearest spellcaster (F,P,R,W)
52. [2] Cure self of a disability (P)
53. [2] Turn undead (P)
54. [2] If there are multiple melee-attackers against self, respond
        to them, preferably to nearest spellcaster (P)
55. [2] Buff self (F,R)
56. [2] Sneak attack (F,R)
57. [2] Melee-attack nearest fighter or rogue (F,R)
58. [2] Use offensive magic against nearest fighter or rogue (P,W)
59. [2] Empty rule (F,P,R,W)
60. [1] Respond to a melee-attacker against self (P,W)
61. [1] Respond to a ranged-attacker against self (P,W)
62. [1] Use offensive magic at an enemy that attacks from a distance
        (P,W)
62. [1] Use offensive magic (P,W)
63. [1] Melee-attack (F,P,R,W)
64. [1] If there are multiple melee-attackers against self, respond
        to them (P)
65. [1] Empty rule (F,P,R,W)
66. [0] Melee-attack (F,P,R,W)
```

Rule 2 forces the agent to use a potion or special ability that enhances its characteristics (which is called 'buffing'). Because of the combat check, this will only be

executed at the start of a fight.

Rule 3 to 7 are 'buffing' rules for priests and wizards. However, the spells used in these rules are unavailable at the experience level of the priest and wizards used in the experiments. Therefore, these rules are effectively empty rules, for the classes that are allowed to use them.

Rule 39 to 58 are extracted without change from the NEVERWINTER NIGHTS game AI version 1.29.

Priests and wizards game-AI scripts contain ten rules extracted from their respective rulebases, while the game-AI scripts of fighters and rogues contain five rules. Rule 9, rule 38, rule 59, and rule 65 are empty rules, that can be selected to make scripts effectively shorter than the number of rules extracted from the rulebase. At the end of a generated script, a call is added to the standard NEVERWINTER NIGHTS game AI. Note that, since version 1.29 and version 1.61 of the standard game AI are different, the effect of this call is dependent on the NEVERWINTER NIGHTS version used.

Appendix C

Wargus Game AI

In Chapter 6, experiments with dynamic scripting in the game WARGUS were discussed. This appendix[1] describes WARGUS and the maps used for the experiments (C.1), the scripting language used to implement game AI (C.2), the static game AI (C.3), the gene types used to design chromosomes (C.4), and the rulebases used to generate successful game AI for the dynamic team (C.5).

C.1 Wargus

WARGUS is a faithful clone of the game WARCRAFT II, released by Blizzard in 1995 (and released again in 1999). WARGUS is built on the open-source game engine STRATAGUS. STRATAGUS was formerly known as FREECRAFT, but for legal reasons the engine has been renamed. STRATAGUS is implemented in C. WARGUS is a game module for STRATAGUS, implemented in the high-level Lua scripting language (Ierusalimschy, de Figueiredo, and Celes, 2003).[2] In the academic community, STRATAGUS is gaining popularity as a research environment for RTS games (Aha and Molineaux, 2004; Marthi, Latham, Russel, and Guestrin, 2004).

The experiments in the WARGUS environment, described in Chapter 6, were performed on two different maps; in the tests where the static game AI employed the 'small balanced tactic' or the 'soldier rush', a small map was used, while in the tests where the static game AI employed the 'large balanced tactic' or the 'knight rush', a large map was used. The two maps are illustrated in Figure C.1. The small map, measuring 64 by 64 cells, is displayed left. The large map, measuring 128 by 128 cells, is displayed right. The black areas on the maps represent water. The mark 'A' indicates the starting base of the dynamic civilisation, and the mark 'B' indicates the starting base of the static civilisation. Note that on the large map the civilisations are far apart, unless they approach each other by sea. However, since naval units were not used during the experiments, the sea route was disabled.

[1] The contents of this appendix are based on the work by Ponsen (2004)

[2] Lua is not an abbreviation. It is the word for 'moon' in Portuguese, and is pronounced 'loo-ah'.

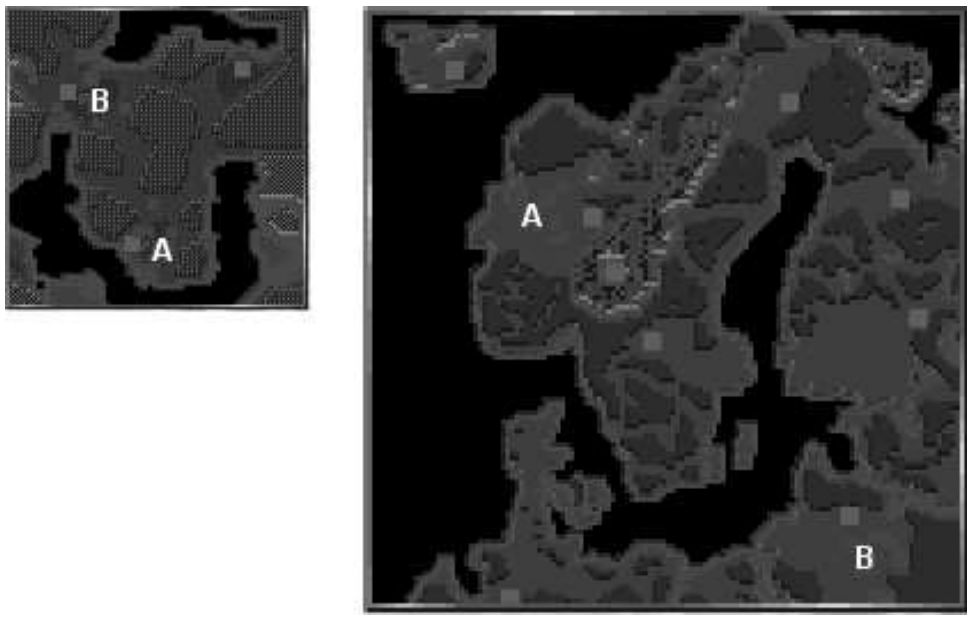

Figure C.1: The two maps used in the WARGUS tests.

C.2 Scripting Language

The WARGUS game AI, implemented in Lua, is based on the concept of 'forces'. A 'force' refers to a group of units, combined in a numbered army. Each unit in the game belongs to a force, and a unit without a force is assigned to a random force automatically. Any commands assigned to a force are assigned to each unit that belongs to the force. WARGUS supports a maximum of ten different forces. A force can be either offensive or defensive. An offensive force will move towards and into the area controlled by the opposing civilisation, attacking enemy units and buildings along the way. A defensive force will stay in the area controlled by its civilisation, responding to enemy attacks. The force numbered zero is always defensive.

A game-AI script for WARGUS is executed sequentially. Each rule in the script is executed (at most) once, starting at the top, and continuing to the bottom, until the game ends.

C.3 Static Tactics

In the WARGUS experiments, the static civilisation uses four different tactics. Two of these tactics, the 'small balanced tactic' and the 'large balanced tactic', use the same game-AI script, but apply it to a small and a large map, respectively. The three game-AI scripts are discussed in the following subsections. Subsection C.3.1 presents

the 'balanced tactic', Subsection C.3.2 presents the 'soldier rush', and Subsection C.3.3 presents the 'knight rush'.

C.3.1 Balanced Tactic

The 'balanced tactic' is an improved variation of the 'land attack' game AI, which was developed by the WARGUS designers. The (rather long) script starts with building a large group of 'workers', whose function is to gather resources and construct buildings. The script then defines a few forces, using them for both attack and defense. When the forces are in place, it constructs all buildings needed to get to state 4 (see Figure 6.2), followed by an extension of the existing forces, followed by the research of all possible weapon and armour upgrades. At that point, the script is able to build fairly strong forces. It mixes the construction of new buildings with extending its existing forces and the creation of new ones, which are used for both offense and defense. If the civilisation manages to get to state 20 (see Figure 6.2), the script continues to build units, which are assigned an offensive or a defensive role, with a ratio of 2 to 1.

C.3.2 Soldier Rush Tactic

The 'soldier rush' tactic aims at overwhelming the enemy with simple soldiers at the start of the game. Since a tactic that is based on the deployment of low-level units works best on a map where the opposing civilisations are close to each other, during the experiments the 'soldier rush' was applied to the small map. The 'soldier rush' script contains the following seventeen steps:

```
 1. Indicate the need for a 'townhall'.
 2. Set the amount of needed 'workers' to 1.
 3. Set the amount of needed 'workers' to 10.
 4. Indicate the need for a 'barracks'.
 5. Build force 0 as two 'soldiers'.
 6. Build force 1 as ten 'soldiers'.
 7. Attack with force 1.
 8. Set the amount of needed 'workers' to 15.
 9. Indicate the need for a 'blacksmith'.
10. Indicate the need for an extra 'barracks'.
11. Research two weapon and two armour upgrades.
12. Build force 0 as four 'soldiers'.
13. Build force 1 as ten 'soldiers'.
14. Attack with force 1.
15. Build force 1 as five 'soldiers'.
16. Attack with force 1.
17. Loop back to step 15.
```

C.3.3 Knight Rush Tactic

The 'knight rush' tactic aims at overwhelming the opposing civilisation with advanced units. The (rather long) script starts similar to the 'soldier rush', but instead of continuously attacking as happens in the 'soldier rush' script after step 12, the 'knight rush' doubles the amount of workers and builds a 'keep', 'stables', a 'lumbermill', and a 'castle', followed by several even more advanced buildings. Then it starts churning out huge forces, consisting of high-level units, and uses them to attack continuously.

C.4 Rule Design

The evolutionary game AI uses a chromosome to specify WARGUS tactics. As detailed in Subsection 6.3.2, a chromosome consists of rule genes. There are four different gene types, namely (i) build genes, (ii) research genes, (iii) economy genes, and (iv) combat genes.

Build genes consist of a rule ID 'B', followed by one numerical parameter, that indicates the type of building to be constructed. The parameter takes an integer value in the range $[1, 12]$. The different parameters for build genes are defined as follows:[3]

```
1 = Townhall     4 = Blacksmith   7 = Castle       10 = Temple
2 = Barracks     5 = Keep         8 = Airport      11 = Guard tower
3 = Lumbermill   6 = Stables      9 = Mage tower   12 = Cannon tower
```

Research genes consist of a rule ID 'R', followed by one numerical parameter, that indicates the type of research to be done. The parameter takes an integer value in the range $[13, 21]$. The different parameters for research genes are defined as follows:

```
13 = Missile upgrade    16 = Catapult upgrade     19 = Mage upgrade 3
14 = Armour upgrade     17 = Mage upgrade 1       20 = Mage upgrade 4
15 = Weapon upgrade     18 = Mage upgrade 2       21 = Mage upgrade 5
```

Economy genes consist of a rule ID 'E', followed by one numerical parameter, that indicates the number of workers to be trained. The parameter takes any positive integer value.

Combat genes consist of a rule ID, consisting of a 'C' and a number, followed by several parameters. The number takes an integer value in the range $[1, 20]$ (corresponding to the twenty possible states, illustrated in Figure 6.2), and determines which parameters the gene has. Combat genes define forces. The first of the parameters is the number of the force to be defined, as an integer value in the range $[0, 9]$. The last of the parameters is the role of the force, namely 'offensive' or 'defensive'.

[3] Note that the 'guard tower' and the 'cannon tower' do not allow new research or the creation of new unit types, therefore they do not spawn state transitions, and thus do not occur in Figure 6.2.

The parameters are unit counts, that specify how many units of a specific type are assigned to the force. For the twenty combat genes, the unit counts are as follows:

```
C01: soldiers
C02: soldiers, shooters
C03: soldiers
C04: soldiers
C05: soldiers, shooters, catapults
C06: soldiers, shooters
C07: soldiers
C08: soldiers
C09: soldiers, shooters, catapults
C10: soldiers, shooters
C11: soldiers, knights
C12: soldiers, shooters, catapults, knights
C13: soldiers, shooters, catapults, knights
C14: soldiers, shooters, catapults, knights, flyers
C15: soldiers, shooters, catapults, knights, mages
C16: soldiers, shooters, catapults, knights
C17: soldiers, shooters, catapults, knights, flyers, mages
C18: soldiers, shooters, catapults, knights, flyers
C19: soldiers, shooters, catapults, knights, mages
C20: soldiers, shooters, catapults, knights, flyers, mages
```

For example, a gene with the value "`C09,1,3,7,2,offensive`" defines force 1 as an offensive force that consists of three soldiers, seven shooters, and two catapults.

C.5 Rulebases

Chapter 6 specified two basic dynamic-scripting rulebases, namely (i) an original rulebase, used in Section 6.2, and (ii) an improved rulebase, used in Section 6.4. From the basic rulebases, separate rulebases for each of the twenty states were constructed, by extracting those rules from the basic rulebases that are applicable in the corresponding states. The two basic rulebases are presented in this section, in Subsections C.5.1 and C.5.2, respectively.

C.5.1 The Original Rulebase

The original WARGUS rulebase, used in Section 6.2, contains fifty rules. The rule specifications use special terms to indicate forces of five different sizes. A 'squadron' is a tiny force (consisting of 2 units), a 'platoon' is a small force (consisting of 4 units), a 'battalion' is a medium-sized force (consisting of 6 units), a 'company' is a large force (consisting of 8 units), and a 'division' is a huge force (consisting of 12 units). The fifty rules are listed below, with a rule number, a rule name, and a short explanation of the rule contents.

1.	Townhall	Construct townhall
2.	Barracks	Construct barracks
3.	Lumbermill	Construct lumbermill
4.	Blacksmith	Construct blacksmith
5.	Keep	Construct keep
6.	Stables	Construct stables
7.	Castle	Construct castle
8.	Airport	Construct airport
9.	Magetower	Construct mage tower
10.	Temple	Construct temple
11.	Guardtower	Construct guard tower
12.	Cannontower	Construct cannon tower
13.	MissileUpgrade	Research better missiles
14.	ArmorUpgrade	Research better armour
15.	WeaponUpgrade	Research better weapons
16.	CatapultUpgrade	Research better catapults
17.	MageUpgrade1	Research mage spell 1
18.	MageUpgrade2	Research mage spell 2
19.	MageUpgrade3	Research mage spell 3
20.	MageUpgrade4	Research mage spell 4
21.	MageUpgrade5	Research mage spell 5
22.	LightWorkers	Train a few new workers
23.	NormalWorkers	Train a several new workers
24.	HeavyWorkers	Train a many new workers
25.	ExtremeWorkers	Train a very many new workers
26.	DefenseSquadron	Define a defensive squadron
27.	DefensePlatoon	Define a defensive platoon
28.	DefenseBattalion	Define a defensive battalion
29.	DefenseCompany	Define a defensive company
30.	DefenseDivision	Define a defensive division
31.	OffenseSquadron	Define an offensive squadron
32.	OffensePlatoon	Define an offensive platoon
33.	OffenseBattalion	Define an offensive battalion
34.	OffenseCompany	Define an offensive company
35.	OffenseDivision	Define an offensive division
36.	SoldiersDefense	Define a defensive force of soldiers
37.	ShootersDefense	Define a defensive force of shooters
38.	CatapultDefense	Define a defensive force of catapults
39.	KnightsDefense	Define a defensive force of knights
40.	MagesDefense	Define a defensive force of mages
41.	SoldiersOffense	Define an offensive force of soldiers
42.	ShootersOffense	Define an offensive force of shooters
43.	CatapultOffense	Define an offensive force of catapults
44.	KnightsOffense	Define an offensive force of knights
45.	MagesOffense	Define an offensive force of mages

```
46. AirDefenseBattalion  Define a defensive air battalion
47. AirDefenseCompany    Define a defensive air company
48. AirOffenseBattalion  Define an offensive air battalion
49. AirOffenseCompany    Define an offensive air company
50. AirOffenseDivision   Define an offensive air division
```

At the end of a game-AI script generated from a rulebase, a continuous loop is added that initiates constant attacks.

C.5.2 The Improved Rulebase

The improved WARGUS rulebase, used in Section 6.4, is based on the original rulebase presented in Subsection C.5.1. The differences are the following.

- Rule 1 has been replaced by a new rule, that defines a defensive force before constructing a new 'townhall'. The reason is that a new townhall will be quickly overrun by enemy units, if it is not defended.
- A new rule has been added, named `AntiSoldierRush`. The rule exists in the rulebase for the state 1. It builds a 'blacksmith' followed by researching two weapon upgrades and two armour upgrades. Then, two offensive forces are defined, one with four soldiers and one with eight soldiers. This rule is meant as a counter-tactic against the 'soldier rush' tactic. When executed, it stems the first wave of 'soldier rush' attacks, and prepares a strong offense with simple units.
- A new rule has been added, named `AntiKnightRush`. The rule exists in the rulebases for states 7 to 11. In state 7 and 8, it builds 'stables'. In state 9 and 10, it builds a 'blacksmith'. In state 11, it builds a 'lumbermill'. In all five states, the construction of the new building is followed by defining two offensive forces consisting of soldiers and knights. The rule aims at quickly switching to a state that allows the construction of 'knights', and exploits this switch by setting up a strong attack using 'knights'.
- A new rule has been added, named `Chromosome`. The rule is a literal copy of a successful chromosome. The rule has implementations for states 3, 4, 8, 12, and 14. The rule is strongly defensive in states 3, 4 and 8, and strongly offensive in states 12 and 14.
- The parameters of rules 26 to 35 have been changed. Four different force sizes have been increased. A 'squadron' now consists of 4 units, a 'platoon' of 6 units, a 'battalion' of 8 units, and a 'company' of 10 units. The size of a 'division' remains at 12 units. Furthermore, the numbers of the units types have been redistributed, to give more weight to 'catapults'.
- Rule 46 to 50, the 'air force' rules, have been removed, to make room for the new rules.

Index

Summary

The behaviour of agents in commercial computer games is determined by so-called 'game AI'. When enhanced with an adaptive mechanism, game AI may learn from its mistakes ('self-correction'), and may change the agents' behaviour in response to unfamiliar situations ('creativity'). Such enhanced game AI is called 'adaptive game AI'. The focus of this thesis is on the design and implementation of machine-learning techniques that can be used to create successful adaptive game AI.

The first chapter provides a motivation for the research, and formulates a problem statement and four research questions. The research is motivated by the fact that game AI in state-of-the-art games lacks sophistication. While the audiovisual qualities of games have undergone considerable improvements in recent years, game AI has been largely neglected by professional game developers. Usually, the suspension of disbelief that modern games attempt to evoke is shattered by the inferior decision-making capabilities of the computer-controlled agents. Adaptive game AI has the potential to extend the time span that a game is challenging for the human player, and to scale the level of difficulty to the human player's level of skill. Implementation of these features may allow adaptive game AI to influence a game's suspension of disbelief positively. So far, academic research in adaptive game AI, small as it is, has focused on simple game AI.

The problem statement derived from the motivation is: *to what extent can machine-learning techniques be used to increase the quality of complex game AI?* To answer the problem statement, four research questions are formulated: (i) to what extent can offline machine-learning techniques be used to increase the effectiveness of game AI? (ii) to what extent can online machine-learning techniques be used to increase the effectiveness of game AI? (iii) to what extent can machine-learning techniques be used to scale the difficulty level of game AI to meet the human player's level of skill? and (iv) how can adaptive game AI be integrated in the game-development process of state-of-the-art games?

The second chapter provides background information. First, it discusses the machine-learning techniques used in the thesis: evolutionary algorithms, artificial neural networks, evolutionary artificial neural networks, evolutionary control, and reinforcement learning. Then, it discusses modern games and state-of-the-art game AI. Finally, it discusses how machine-learning techniques can be applied to game AI, and gives an overview of related research in this area. The three ways by which machine learning can be applied to game AI are (i) offline learning, (ii) supervised

learning (which is excluded from this thesis), and (iii) online learning. Offline adaptive game AI is game AI that adapts using self-play, typically during the 'quality assurance' phase of game development. Online adaptive game AI is game AI that adapts while the game is being played by a human player. Online adaptive game AI must meet four computational and four functional requirements to be applicable in practice. The four computational requirements are (i) speed, (ii) effectiveness, (iii) robustness, and (iv) efficiency. The four functional requirements are (i) clarity, (ii) variety, (iii) consistency, and (iv) scalability.

The third chapter discusses how to evolve successful agent controllers in game-like environments. When evolving agent controllers, the evolutionary algorithm tends to seek solutions in the search space in the neighbourhood of solutions to easy problem instances. Consequently, the solutions found tend to work well with easy instances, but give inferior results with hard instances. This is called 'the problem of hard instances'. To deal with this problem, a novel evolutionary algorithm is introduced, called the Doping-driven Evolutionary Control Algorithm (DECA). DECA 'dopes' the initial population of potential solutions with a very good solution to a single hard instance. Through experiments with a box-pushing task and with a food-gathering task, the chapter empirically shows that DECA evolves agent controllers that are significantly more effective than agent controllers evolved with a 'regular' evolutionary algorithm.

The fourth chapter explores evolutionary game AI, which is game AI that employs evolutionary algorithms to adapt. The first part of the chapter discusses offline evolutionary game AI. By an experiment that controls the actions of a spaceship in a strategy game with a neural network, it shows that offline evolutionary game AI can be successful in detecting exploits, and in discovering new tactics. However, the first part concludes with the observation that a neural network is not a suitable learning structure for game AI. The second part discusses online evolutionary game AI. By an experiment that evolves team behaviour in the capture-the-flag mode of the action game QUAKE III ARENA, it shows that online evolutionary game AI can be used to create successful tactics. However, it is concluded that online evolutionary game AI is only reasonably efficient if the search space is small.

The fifth chapter discusses a novel technique for online adaptive game AI called 'dynamic scripting'. Dynamic scripting maintains game-domain knowledge in the form of rules in an adaptive rulebase. Each rule has a weight attached to it, which determines the probability that the associated rule is selected for a game-AI script. The weights adapt automatically to reflect the success or failure of the game AI as observed in the game. The chapter shows that dynamic scripting meets by design all four computational requirements, and two of the four functional requirements (namely clarity and variety). The chapter then explores (i) outlier-reduction enhancements to dynamic scripting to allow it to meet the requirement of consistency, and (ii) difficulty-scaling enhancements to allow it to meet the requirement of scalability. With 'penalty balancing' as an outlier-reduction enhancement, and 'top culling' as a difficulty-scaling enhancement, dynamic scripting meets all four computational and all four functional requirements. Therefore, it is concluded that dynamic scripting can be applied in practice. The conclusion is supported by the

successful implementation of dynamic scripting in the state-of-the-art roleplaying game NEVERWINTER NIGHTS.

The sixth chapter discusses how adaptive game AI can be integrated in professional game development. It shows that game developers and publishers will not hesitate to use offline adaptive game AI when they believe that they can benefit from it. However, at present they are still suspicious of online adaptive game AI, and need to be convinced of its reliability to start considering applying it in their games. The reliability of online adaptive game AI can be improved by using offline adaptive game AI to discover new domain knowledge. A three-step procedure to execute this improvement is illustrated by an experiment with the game AI in the real-time strategy game WARGUS. The experiment shows that a dynamic-scripting rulebase for WARGUS can be improved by using offline evolutionary game AI to design counter-tactics against 'super-tactics', which are quite difficult to defeat. The chapter ends by discussing some generalisation issues, and by providing arguments that support the conjecture that adaptive game AI is beneficial to the entertainment value derived from games.

The seventh chapter returns to the problem statement and research questions. The answers to the research questions are all given above. They provide the following, four-part answer to the problem statement:

- reliability of online adaptive game AI is guaranteed if it meets the four computational and four functional requirements;
- offline machine-learning techniques can be used during the 'quality assurance' phase of game development to increase the effectiveness of game AI by (i) detecting exploits, (ii) suggesting new tactics, and (iii) increasing the reliability of online adaptive game AI by improving the quality of the domain knowledge used;
- after a game's release, online machine-learning techniques can (i) improve the effectiveness of game AI, and (ii) scale the difficulty level of game AI to match the playing strength of the human player; and
- game developers and publishers will consider using online adaptive game AI when they are convinced that it is reliable.

The consensus amongst game developers and publishers seems to be that adaptive game AI is something to be avoided. Still, adaptive game AI is an essential element for truly believable characters in computer games. This thesis shows that adaptive game AI can be successful, and be reliable, both in offline and online implementations. The question is therefore not if, but when adaptive game AI will become a standard element of games.

Samenvatting

Het gedrag van agenten in commerciële computer games[1] wordt bepaald door zogeheten *game AI*. Als game AI wordt uitgebreid met een adaptief mechanisme, kan ze leren van de eigen fouten (zelf-correctie), en het gedrag van de agenten aanpassen aan ongewone situaties (creativiteit). Een dergelijke game AI wordt *adaptive game AI* genoemd. Dit proefschrift focust op het ontwerp en de implementatie van *machine-learning* technieken die successvolle adaptive game AI mogelijk maken.

Het eerste hoofdstuk geeft een motivatie voor het onderzoek, en formuleert een probleemstelling en vier onderzoeksvragen. Het onderzoek wordt sterk gemotiveerd door een gebrek aan raffinement bij de game AI van moderne games. Terwijl de audiovisuele kwaliteiten van games de laatste jaren met sprongen vooruit zijn gegaan, hebben professionele game-ontwikkelaars de game AI grotendeels genegeerd. Game-ontwikkelaars trachten bij spelers de beleving op te roepen dat de wereld voorgesteld in een game werkelijkheid is (dit wordt aangeduid met de term 'immersie'). Deze beleving wordt meestal teniet gedaan door het inferieure gedrag van de computergestuurde agenten. Adaptive game AI heeft de mogelijkheid de tijdsduur te verlengen dat een game uitdagend blijft voor een menselijke speler. Daarnaast kan ze de moeilijkheidsgraad van een game automatisch aanpassen aan de speelsterkte van de menselijke speler. Implementatie van deze eigenschappen kan ervoor zorgen dat adaptive game AI het gevoel van immersie bij de menselijke speler versterkt. Tot voor kort was academisch onderzoek naar adaptive game AI beperkt tot de game AI voor eenvoudige games.

De probleemstelling, direct afgeleid uit de bovengeschetste motivatie, luidt: *In hoeverre is het mogelijk om machine-learning technieken te gebruiken om de kwaliteit van complexe game AI te verhogen?* Om deze vraag te beantwoorden, zijn vier onderzoeksvragen geformuleerd: (i) In hoeverre is het mogelijk om *offline* machine-learning technieken te gebruiken om de effectiviteit van game AI te vergroten? (ii) In hoeverre is het mogelijk om *online* machine-learning technieken te gebruiken om de effectiviteit van game AI te vergroten? (iii) In hoeverre kunnen machine-learning technieken gebruikt worden om de moeilijkheidsgraad van game AI te schalen naar de speelsterkte van de menselijke speler? en (iv) Hoe kan adaptive game AI worden geïntegreerd in het proces van game-ontwikkeling van moderne games?

[1] De Nederlandse vertaling van 'computer games' is 'computerspelen', maar in het dagelijks gebruik geniet de Engelse benaming de voorkeur. Daarnaast worden commerciële computer games meestal aangeduid met de verkorte term 'games'. Dit gebruik is in het proefschrift overgenomen.

Het tweede hoofdstuk geeft enige achtergrondinformatie bij het onderzoek. Het hoofdstuk begint met een bespreking van de machine-learning technieken die in het proefschrift gebruikt worden: evolutionaire algoritmen, neurale netwerken, evolutionaire neurale netwerken, evolutionaire besturing, en reïnforcement leren. Daarna volgt een bespreking van moderne games en hun game AI. Tenslotte bespreekt het hoofdstuk de toepassing van machine-learning technieken op game AI, en geeft het een overzicht van aanpalend onderzoek op dit gebied. De drie manieren waarop machine learning kan worden toegepast op game AI zijn: (i) *offline learning*, (ii) *supervised learning* (die niet wordt behandeld in dit proefschrift), en (iii) *online learning*. Offline adaptive game AI is game AI die zich aanpast door tegen zichzelf te spelen. Gewoonlijk gebeurt dit tijdens de testfase van een game. Online adaptive game AI is game AI die zich aanpast tijdens het spelen van een game door een mens. Om praktisch toepasbaar te zijn, moet online adaptive game AI voldoen aan vier computationele eisen, en aan vier functionele eisen. De vier computationele eisen zijn: (i) snelheid, (ii) effectiviteit, (iii) robuustheid, en (iv) efficiëntie. De vier functionele eisen zijn: (i) helderheid, (ii) variëteit, (iii) consistentie, en (iv) schaalbaarheid.

Het derde hoofdstuk bespreekt hoe succesvolle agent-besturing geëvolueerd kan worden in een spel-achtige omgeving. Wanneer agentbesturing geëvolueerd wordt, zoekt een evolutionair algoritme over het algemeen in de zoekruimte een oplossing in de buurt van oplossingen voor een eenvoudige probleem-instantie. Het gevolg is dat de uiteindelijke oplossing vaak goed werkt op eenvoudige instanties, maar slecht op moeilijke instanties. Dit heet 'het probleem van de moeilijke instanties'. Om dit probleem op te lossen, introduceert het hoofdstuk een nieuw evolutionair algoritme dat het *Doping-driven Evolutionary Control Algorithm* (DECA) wordt genoemd. DECA voorziet een initiële populatie van mogelijke oplossingen van een zeer goede oplossing voor een moeilijke instantie. Met behulp van twee experimenten met ieder een verschillende taak (namelijk het verplaatsen van een doos door een robot, en het vergaren van voedsel door een agent) toont het hoofdstuk aan dat DECA agentbesturingen evolueert die significant effectiever zijn dan agentbesturingen die geëvolueerd zijn met reguliere evolutionaire algoritmen.

Het vierde hoofdstuk handelt over evolutionaire game AI. Dit is game AI die zich aanpast middels evolutionaire algoritmen. Het eerste deel van het hoofdstuk bespreekt offline evolutionaire game AI. Met behulp van een experiment waarbij een neuraal netwerk wordt geëvolueerd voor de aansturing van een ruimteschip in een strategisch spel, wordt aangetoond dat offline evolutionaire game AI succesvol kan zijn in het ontdekken van exploiteerbare zwakheden, en van nieuwe tactieken. Niettemin wordt geconcludeerd dat neurale netwerken niet bijster geschikt zijn voor het leren van game AI. Het tweede deel bespreekt online evolutionaire game AI. Met behulp van een experiment, waarbij groepsgedrag wordt geëvolueerd voor het vlagveroveren in het actie-spel Quake III Arena, wordt aangetoond dat online evolutionaire game AI gebruikt kan worden voor het genereren van succesvolle tactieken. Er wordt echter geconcludeerd dat online evolutionaire game AI slechts redelijk efficiënt is indien de zoekruimte klein is.

Het vijfde hoofdstuk bespreekt een nieuwe techniek voor online adaptive game AI, *dynamic scripting* genaamd. Dynamic scripting onderhoudt domeinkennis over

een game in de vorm van regels in een adaptieve kennisbank. Elke regel is voorzien van een gewicht, dat de kans aangeeft dat de geassocieerde regel gebruikt wordt in een game-AI script. De gewichten passen zich automatisch aan naar aanleiding van het geobserveerde succes of falen van de game AI tijdens het spelen. Het hoofdstuk toont aan dat dynamic scripting voldoet aan alle vier de computationele eisen, en aan twee van de vier functionele eisen (namelijk helderheid en variëteit). Daarna wordt in het hoofdstuk onderzoek gedaan naar maatregelen ten behoeve van de bevordering van consistentie, en van de schaalbaarheid. Met *penalty balancing* als consistentie-bevorderende maatregel, en *top culling* als schaalbaarheids-maatregel, voldoet dynamic scripting aan alle vier de computationele, en alle vier de functionele eisen. Er wordt daarom geconcludeerd dat dynamic scripting in de praktijk kan worden toegepast. Deze conclusie wordt gestaafd door de succesvolle implementatie van dynamic scripting in het moderne *computer roleplaying game* NEVERWINTER NIGHTS.

Het zesde hoofdstuk bespreekt hoe adaptive game AI kan worden geïntegreerd in de praktijk van game-ontwikkeling. Het hoofdstuk laat zien dat ontwikkelaars en uitgevers van games niet zullen aarzelen om offline adaptive game AI toe te passen wanneer ze denken daarmee winst te kunnen behalen. Op dit moment staan ze echter wantrouwend tegenover online adaptive game AI. Ze zullen overtuigd moeten worden van de betrouwbaarheid van online adaptive game AI, voordat ze zullen overwegen het toe te passen in hun games. De betrouwbaarheid van online adaptive game AI kan worden vergroot door offline adaptive game AI in te zetten voor het ontdekken van nieuwe domeinkennis. Een drie-stappen procedure die dit bewerkstelligt, wordt geïllustreerd aan de hand van een experiment met adaptive game AI in het *real-time strategy game* WARGUS. Het experiment toont aan dat een dynamic-scripting kennisbank voor WARGUS verbeterd kan worden door offline evolutionaire game AI te gebruiken voor de weerlegging van 'super-tactieken', die slechts met veel moeite verslagen kunnen worden. Het hoofdstuk sluit af met een discussie over generalisatie-mogelijkheden, en het geven van een argument waarom adaptive game AI positief kan bijdragen aan de entertainment-waarde die mensen ervaren bij het spelen van een game.

Het zevende hoofdstuk keert terug naar de probleemstelling en onderzoeksvragen. De antwoorden op de onderzoeksvragen zijn hierboven gegeven. Zij leiden direct tot het volgende antwoord op de probleemstelling, dat bestaat uit vier delen:

- De betrouwbaarheid van online adaptive game AI is gegarandeerd als de game AI voldoet aan de vier computationele eisen en aan de vier functionele eisen.
- Offline machine-learning technieken kunnen worden gebruikt tijdens de test-fase van een game, om de effectiviteit van de game AI te vergroten door (i) zwakheden bloot te leggen, (ii) nieuwe tactieken te suggereren, en (iii) de betrouwbaarheid van online adaptive game AI te vergroten door de kwaliteit van de domeinkennis te verbeteren.
- Nadat een game op de markt is gekomen, kunnen online machine-learning technieken gebruikt worden om (i) de effectiviteit van game AI te vergroten,

en (ii) de moeilijkheidsgraad van de game AI te schalen naar de speelsterkte van de menselijke speler.

- Game-ontwikkelaars en uitgevers zullen het gebruik van online adaptive game AI in overweging willen nemen als ze overtuigd zijn van de betrouwbaarheid ervan.

Onder game-ontwikkelaars en uitgevers lijkt de consensus te zijn dat adaptive game AI vermeden dient te worden. Toch is adaptive game AI een essentieel element voor de creatie van werkelijk geloofwaardige personages in een game. Dit proefschrift toont aan dat adaptive game AI succesvol en betrouwbaar kan zijn, in zowel offline als online implementaties. De vraag is daarom niet zozeer of, maar wanneer adaptive game AI een standaard element in games zal zijn.

Curriculum Vitae

Men do not quit playing because they grow old;
they grow old because they quit playing.
— Justice Oliver Wendell Holmes (1809–1894).

Pieter Spronck was born in Maastricht, on June 6, 1963. He attended secondary school at the Scholengemeenschap Jeanne d'Arc in Maastricht, where he received his Gymnasium diploma in 1981. He studied mathematics at the Utrecht University, but left without graduating. He became a programmer with Coss Holland, Nieuwegein in 1987. In 1990 he started a part-time study of computer science at the Delft University of Technology, while keeping a full-time day-job. In 1993 he switched jobs, and became a designer and programmer of electronic-banking systems at Credit Lyonnais Bank, Rotterdam. He received his master's degree cum laude in 1996, with a thesis on the use of genetic algorithms to design neural controllers. In 1997 he joined the Netherlands Organisation for Applied Scientific Research (TNO), Delft, where he designed and built knowledge-based systems. In pursuit of a Ph.D., in 2001 he was appointed at the Institute for Knowledge and Agent Technology (IKAT) of the Universiteit Maastricht. There he investigates adaptive behaviour in games, and is involved with contract research. He lives in Maastricht with his wife, Muriël, and their daughter, Myrthe. His homepage can be found at `www.spronck.net`.

Pieter admits to be something of a gaming geek. His favourite computer games include M.U.L.E. (1983), Elite (1985), Quest of the Avatar (1985), Martian Dreams (1991), The Fate of Atlantis (1992), The Stygian Abyss (1992), System Shock (1994), Master of Magic (1995), The Curse of Monkey Island (1997), Grim Fandango (1998), Planescape: Torment (1999), Shadows of Amn (2000), Deus Ex (2000), The Metal Age (2000), Morrowind (2002), Knights of the Old Republic (2003), and Deadly Shadows (2004). He is fiercely proud of the fact that, in his early twenties, he solved the original Zork trilogy unassisted. It took him almost two years.

SIKS Dissertation Series

1998

1 Johan van den Akker (CWI[1]) *DEGAS – An Active, Temporal Database of Autonomous Objects*

2 Floris Wiesman (UM) *Information Retrieval by Graphically Browsing Meta-Information*

3 Ans Steuten (TUD) *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective*

4 Dennis Breuker (UM) *Memory versus Search in Games*

5 Eduard W. Oskamp (RUL) *Computerondersteuning bij Straftoemeting*

1999

1 Mark Sloof (VU) *Physiology of Quality Change Modelling; Automated Modelling of Quality Change of Agricultural Products*

2 Rob Potharst (EUR) *Classification using Decision Trees and Neural Nets*

3 Don Beal (UM) *The Nature of Minimax Search*

4 Jacques Penders (UM) *The Practical Art of Moving Physical Objects*

5 Aldo de Moor (KUB) *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*

6 Niek J.E. Wijngaards (VU) *Re-Design of Compositional Systems*

7 David Spelt (UT) *Verification Support for Object Database Design*

8 Jacques H.J. Lenting (UM) *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation*

2000

1 Frank Niessink (VU) *Perspectives on Improving Software Maintenance*

[1] Abbreviations: SIKS – Dutch Research School for Information and Knowledge Systems; CWI – Centrum voor Wiskunde en Informatica, Amsterdam; EUR – Erasmus Universiteit, Rotterdam; KUB – Katholieke Universiteit Brabant, Tilburg; KUN – Katholieke Universiteit Nijmegen; RUL – Rijksuniversiteit Leiden; RUN – Radboud Universiteit Nijmegen; TUD – Technische Universiteit Delft; TU/e – Technische Universiteit Eindhoven; UL – Universiteit Leiden; UM – Universiteit Maastricht; UT – Universiteit Twente, Enschede; UU – Universiteit Utrecht; UvA – Universiteit van Amsterdam; UvT – Universiteit van Tilburg; VU – Vrije Universiteit, Amsterdam.

2 Koen Holtman (TU/e) *Prototyping of CMS Storage Management*

3 Carolien M.T. Metselaar (UvA) *Sociaal-organisatorische Gevolgen van Kennistechnologie; een Procesbenadering en Actorperspectief*

4 Geert de Haan (VU) *ETAG, A Formal Model of Competence Knowledge for User Interface Design*

5 Ruud van der Pol (UM) *Knowledge-Based Query Formulation in Information Retrieval*

6 Rogier van Eijk (UU) *Programming Languages for Agent Communication*

7 Niels Peek (UU) *Decision-Theoretic Planning of Clinical Patient Management*

8 Veerle Coupé (EUR) *Sensitivity Analyis of Decision-Theoretic Networks*

9 Florian Waas (CWI) *Principles of Probabilistic Query Optimization*

10 Niels Nes (CWI) *Image Database Management System Design Considerations, Algorithms and Architecture*

11 Jonas Karlsson (CWI) *Scalable Distributed Data Structures for Database Management*

2001

1 Silja Renooij (UU) *Qualitative Approaches to Quantifying Probabilistic Networks*

2 Koen Hindriks (UU) *Agent Programming Languages: Programming with Mental Models*

3 Maarten van Someren (UvA) *Learning as Problem Solving*

4 Evgueni Smirnov (UM) *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets*

5 Jacco van Ossenbruggen (VU) *Processing Structured Hypermedia: A Matter of Style*

6 Martijn van Welie (VU) *Task-Based User Interface Design*

7 Bastiaan Schonhage (VU) *Diva: Architectural Perspectives on Information Visualization*

8 Pascal van Eck (VU) *A Compositional Semantic Structure for Multi-Agent Systems Dynamics*

9 Pieter Jan 't Hoen (RUL) *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes*

10 Maarten Sierhuis (UvA) *Modeling and Simulating Work Practice BRAHMS: a Multiagent Modeling and Simulation Language for Work Practice Analysis and Design*

11 Tom M. van Engers (VU) *Knowledge Management: The Role of Mental Models in Business Systems Design*

2002

1 Nico Lassing (VU) *Architecture-Level Modifiability Analysis*

2 Roelof van Zwol (UT) *Modelling and Searching Web-based Document Collections*

3 Henk Ernst Blok (UT) *Database Optimization Aspects for Information Retrieval*

4 Juan Roberto Castelo Valdueza (UU) *The Discrete Acyclic Digraph Markov Model in Data Mining*

5 Radu Serban (VU) *The Private Cyberspace Modeling Electronic Environments Inhabited by Privacy-Concerned Agents*

6 Laurens Mommers (UL) *Applied Legal Epistemology; Building a Knowledge-based Ontology of the Legal Domain*

7 Peter Boncz (CWI) *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*

8 Jaap Gordijn (VU) *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas*

9 Willem-Jan van den Heuvel (KUB) *Integrating Modern Business Applications with Objectified Legacy Systems*

10 Brian Sheppard (UM) *Towards Perfect Play of Scrabble*

11 Wouter C.A. Wijngaards (VU) *Agent Based Modelling of Dynamics: Biological and Organisational Applications*

12 Albrecht Schmidt (UvA) *Processing XML in Database Systems*

13 Hongjing Wu (TU/e) *A Reference Architecture for Adaptive Hypermedia Applications*

14 Wieke de Vries (UU) *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems*

15 Rik Eshuis (UT) *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*

16 Pieter van Langen (VU) *The Anatomy of Design: Foundations, Models and Applications*

17 Stefan Manegold (UvA) *Understanding, Modeling, and Improving Main-Memory Database Performance*

2003

1 Heiner Stuckenschmidt (VU) *Ontology-Based Information Sharing in Weakly Structured Environments*

2 Jan Broersen (VU) *Modal Action Logics for Reasoning About Reactive Systems*

3 Martijn Schuemie (TUD) *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy*

4 Petkovic (UT) *Content-Based Video Retrieval Supported by Database Technology*

5 Jos Lehmann (UvA) *Causation in Artificial Intelligence and Law – A Modelling Approach*

6 Boris van Schooten (UT) *Development and Specification of Virtual Environments*

7 Machiel Jansen (UvA) *Formal Explorations of Knowledge Intensive Tasks*

8 Yong-Ping Ran (UM) *Repair-Based Scheduling*

9 Rens Kortmann (UM) *The Resolution of Visually Guided Behaviour*

10 Andreas Lincke (UT) *Electronic Business Negotiation: Some Experimental Studies on the Interaction between Medium, Innovation Context and Cult*

11 Simon Keizer (UT) *Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks*

12 Roeland Ordelman (UT) *Dutch Speech Recognition in Multimedia Information Retrieval*

13 Jeroen Donkers (UM) *Nosce Hostem – Searching with Opponent Models*

14 Stijn Hoppenbrouwers (KUN) *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations*

15 Mathijs de Weerdt (TUD) *Plan Merging in Multi-Agent Systems*

16 Menzo Windhouwer (CWI) *Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouse*

17 David Jansen (UT) *Extensions of Statecharts with Probability, Time, and Stochastic Timing*

18 Levente Kocsis (UM) *Learning Search Decisions*

2004

1 Virginia Dignum (UU) *A Model for Organizational Interaction: Based on Agents, Founded in Logic*

2 Lai Xu (UvT) *Monitoring Multi-party Contracts for E-business*

3 Perry Groot (VU) *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving*

4 Chris van Aart (UvA) *Organizational Principles for Multi-Agent Architectures*

5 Viara Popova (EUR) *Knowledge Discovery and Monotonicity*

6 Bart-Jan Hommes (TUD) *The Evaluation of Business Process Modeling Techniques*

7 Elise Boltjes (UM) *Voorbeeld$_{IG}$ Onderwijs; Voorbeeldgestuurd Onderwijs, een Opstap naar Abstract Denken, vooral voor Meisjes*

8 Joop Verbeek (UM) *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale Politiële Gegevensuitwisseling en Digitale Expertise*

9 Martin Caminada (VU) *For the Sake of the Argument; Explorations into Argument-based Reasoning*

10 Suzanne Kabel (UvA) *Knowledge-rich Indexing of Learning-objects*

11 Michel Klein (VU) *Change Management for Distributed Ontologies*

12 The Duy Bui (UT) *Creating Emotions and Facial Expressions for Embodied Agents*

13 Wojciech Jamroga (UT) *Using Multiple Models of Reality: On Agents who Know how to Play*

14 Paul Harrenstein (UU) *Logic in Conflict. Logical Explorations in Strategic Equilibrium*

15 Arno Knobbe (UU) *Multi-Relational Data Mining*

16 Federico Divina (VU) *Hybrid Genetic Relational Search for Inductive Learning*

17 Mark Winands (UM) *Informed Search in Complex Games*

18 Vania Bessa Machado (UvA) *Supporting the Construction of Qualitative Knowledge Models*

19 Thijs Westerveld (UT) *Using Generative Probabilistic Models for Multimedia Retrieval*

20 Madelon Evers (Nyenrode) *Learning from Design: Facilitating Multidisciplinary Design Teams*

2005

1 Floor Verdenius (UvA) *Methodological Aspects of Designing Induction-Based Applications*

2 Erik van der Werf (UM) *AI Techniques for the Game of Go*

3 Franc Grootjen (RUN) *A Pragmatic Approach to the Conceptualisation of Language*

4 Nirvana Meratnia (UT) *Towards Database Support for Moving Object Data*

5 Gabriel Infante-Lopez (UvA) *Two-Level Probabilistic Grammars for Natural Language Parsing*

6 Pieter Spronck (UM) *Adaptive Game AI*